Validation Sequence Optimization: A Theoretical Approach

Gediminas Adomavicius • Alexander Tuzhilin

Department of Information and Decision Sciences, Carlson School of Management, University of Minnesota, 321 19th Avenue South, Minneapolis, Minnesota 55455, USA
Department of Information, Operations, and Management Sciences, Stern School of Business, New York University, 44 West 4th Street, New York, New York 10012, USA
gedas@umn.edu • atuzhili@stern.nyu.edu

The need to validate large amounts of data with the help of the domain expert arises naturally in many data-intensive applications, including a variety of data mining, data stream, and database-related applications. This paper presents a general validation approach that generalizes different expert-driven validation methods developed for specialized validation problems. In particular, we model the validation process as a sequence of validation operators, explore various properties of such sequences, and present theoretical results that provide for better understanding of the validation process. We also address the problem of selecting the best validation sequence among the class of equivalent sequence permutations. We demonstrate that this optimization problem is NP-hard and present two heuristic algorithms for improving validation sequences.

(Validation; Validation Operators; Validation Sequences; Sequence Optimization; Computational Complexity; Heuristic Algorithms; Data Mining)

1. Introduction and Motivation

This paper addresses a problem of validating large amounts of data and information generated in various applications. This problem is especially common in data mining, where the need to validate the results of the data mining methods during the post-analysis stage of the knowledge discovery process arises naturally in a number of situations. For example, [3] describes the process of validating user profiles in personalization applications, [23] describes methods for validating biological relationships in bioinformatics applications, and [14, 17, 18, 1] describes the general issues of validating frequent itemsets and association rules, discovered by the Apriori data mining algorithm [5]. In particular, the personalization
problem [3] can generate a very large number of discovered rules (such as “when a person travels on business to Los Angeles, she tends to stay in expensive hotels there”) that can be measured in hundreds of millions for large-scale CRM applications. The goal of the validation stage of the data mining process is to determine which of these rules are truly interesting and which ones can be discarded as obvious or irrelevant for a particular application. Without good validation methods deployed in the post-analysis stage of the knowledge discovery process, it is impossible to determine the quality of the results generated during the data mining stage [8, 10]. Therefore, the whole knowledge discovery process crucially depends on the quality of the validation step.

While data mining applications were the ones that provided the primary motivation for this research, the validation problem can occur in other types of applications as well. For example, Babu et al. [6] consider the problem of efficient validation of data streams, where a continuous stream of data is processed by a set of commutative filters. This problem can be critical in many “data stream” applications, i.e., applications where data is updated continuously and needs to be processed in real-time, such as network monitoring, sensor processing, telecommunications fraud detection, and financial stock monitoring [6]. Still another category of applications relevant to this work is validation of the transactional data using database queries. For instance, any application where records in a relational table have to be classified into several categories (i.e., validated) by the domain expert would fit into such category such application, e.g., student admission process into colleges, where all candidates in a college database are either accepted, rejected, or put on a waiting list based on a variety of different criteria (represented as database queries).

Many authors advocate the direct involvement of the user (e.g., domain expert) in the process of validation, and the rule validation problem in the post-analysis stage of the knowledge discovery process has been addressed before [14, 17, 22, 13, 18]. In addition, it has been observed by several researchers, e.g., [8, 10, 21, 19, 16, 2, 20], that knowledge discovery should be an iterative and interactive process that involves an explicit participation of the domain expert. Although particular validation methods described in [14, 22, 13, 3, 23, 1] are different and were developed for different types of applications, they have certain common ideas that go beyond data mining applications and can be applied to other important IT problems, such as processing of data streams [6] or validation of transactional data using database queries. In this paper, we present a general theory behind different expert-driven validation methods developed for particular validation problems described above. In par-
ticular, we model the validation process as a sequence of validation operators. Validation operators were introduced in the context of the data mining rule validation in [2, 3, 23], where we also studied the methods for generating sequences of these operators.

In this paper we assume that a sequence of validation operators is already defined (typically with the help of a domain expert) and we study the optimization problem of how to replace the initially specified sequence with an equivalent but more efficient sequence of validation operators. The ability to produce more efficient validation sequences is important in data mining applications for the following reasons. First, in many applications, new data keeps arriving over time and it affects previously generated patterns, e.g., these patterns may no longer hold in light of the newly available data. In addition, the new data may also facilitate the discovery of completely new patterns. As the result, it is necessary to re-evaluate and re-validate the data mining results. In large applications having frequently changing data, this re-evaluation and re-validation process can be computationally very intensive and would not fit within a specified timeframe and, therefore, it is crucial to validate sequences in the most efficient manner. Second, validation constitutes an iterative and interactive process, where the domain expert usually waits for the results of the previous validation step. Therefore, generation of more efficient sequences would reduce waiting time of the domain expert. This is really crucial in large-scale applications dealing with millions of generated patterns where excessive waiting time can easily make a validation application impractical. Moreover, the need for efficient validation exists not only in data mining applications, but in other types of applications as well. For example, the above-mentioned work on data streams [6] not only considers the problem of data stream processing, but also investigates the possibilities for optimizing the sequences of pipelined filters. This problem is very important for data streams because the stream validation process occurs in real-time, and it is crucial to validate these streams faster than the data arrives.

To keep the problem applicable to a broad set of applications, including validation of data streams and transactional data in databases, we study validation of such sequences in a setting that is more general and abstract than a specific application of data mining rule validation. We also study various properties of these sequences, including sequence equivalence, sequence permutation and sequence optimality properties. We also present several theoretical results providing for a better understanding of the validation process. Finally, we address the problem of selecting the best validation sequence among the class of equivalent sequences. We show that this sequence optimization problem is NP-hard. We
also present two heuristic algorithms for improving validation sequence performance.

Performance improvements of an optimized sequence depend significantly on the application and on how well the domain expert specified the initial validation sequence. If the expert did a perfect job and specified an optimal sequence, then, certainly, the proposed optimization methods cannot improve this sequence at all. On the other hand, if the domain expert initially specified a poorly performing sequence, then the proposed methods can provide significant performance improvements. One of the strengths of our approach is that it makes the validation process adaptive, i.e., if the characteristics of the data to be validated change over time, the validation sequence performance may also change. The proposed sequence optimization algorithm can be applied periodically to make sure that the performance of the validation sequence does not deteriorate over time as data characteristics change.

The strength of our work lies not only in building the theory behind various specific validation methods described in [14, 22, 13, 3, 24, 23, 1], but also in generalizing and abstracting the specific approaches presented in these references. In fact, the approach described in this paper is applicable to any data validation problem having the following properties:

- These problems generate very large numbers of data points, such as data mining rules or data streams, that cannot be validated individually. For example, personalization applications can generate millions of rules comprising profiles of individual customers [3].

- Generated data points can belong to different categories and need to be classified using several labels. For example, user profiling methods often use data mining methods that can generate many spurious, obvious or irrelevant rules [3]. Such rules have to be labeled as “bad” by the domain expert and should be separated from the rules labeled as “good” or “undetermined” and removed from user profiles.

- Such problems are knowledge-intensive and require direct involvement of the domain expert in the validation process since data mining methods cannot leverage this knowledge as well as the domain expert would do.

The rest of the paper is organized as follows. Section 2 presents a general approach to validation that is based on sequences of validation operators. Section 3 introduces the problem of sequence optimization, and Section 4 presents a permutation-based approach to address this problem and describes various theoretical properties of validation sequence
permutations. Section 5 analyzes the computational complexity of the validation sequence optimization problem. Section 6 describes the proposed heuristic algorithms for improving validation sequences, and some concluding remarks and the discussion on future research are presented in Section 7. Finally, proofs of the main theoretical results are presented in the Appendix A.

2. General Validation Problem

The general validation problem can formally be stated as follows. Assume that we have a finite set $\mathcal{E}$ that contains all possible data points that may potentially need to be validated by the domain expert. Then dataset $D$ is simply some subset of all possible data, i.e., $D \subseteq \mathcal{E}$.

The domain expert “validates” dataset $D$ by assigning labels from the label set $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ to each input element $e \in D$. More formally, the goal of the validation process is to split the input set $D$ into $n + 1$ pairwise disjoint sets $V_1, V_2, \ldots, V_n,$ and $U$, where each $V_i$ represents the subset of $D$ that was labeled with $L_i$, and $U$ denotes the subset of $D$ that remains unlabeled after the validation process (i.e., some input elements may remain unvalidated).

Since dataset $D$ may contain a very large number of data elements, it is not feasible for the domain expert to validate all the elements in $D$ manually, i.e., by inspecting and labeling them on the one-by-one basis. To make the expert-driven validation feasible, we propose to use validation operators, i.e., methods that allow the expert to validate multiple elements of $D$ at a time. This is achieved by letting the domain expert specify logical predicates that label a class of data elements with a particular label.

Formally, let’s denote $\mathcal{P}$ to be the set of all possible predicates for validating input data from $\mathcal{E}$, i.e., let $\mathcal{P}$ contain all predicates $p$ of the form

$$p : \mathcal{E} \longrightarrow \{\text{True}, \text{False}\}. \quad (1)$$

Then, the validation operator is defined as follows.

**Definition 1 (Validation Operator)** A tuple $(l, p)$, where $l \in \mathcal{L}$ and $p \in \mathcal{P}$, is called a validation operator.

In other words, for an unvalidated dataset $D$ and an expert-specified validation operator $o = (l, p)$, all data points $e \in D$ for which $p(e) = \text{True}$ are labeled with $l$ and are considered
validated. Data points \( e \in D \) for which \( p(e) = \text{False} \) remain unvalidated. Therefore, validation is an iterative process, where in each iteration the domain expert can specify a new validation operator that validates still another portion of yet unvalidated part of dataset \( D \). Thus, the validation process can be described as a sequence of validation operators.

**Definition 2 (Validation Sequence)** A sequence of validation operators is called a validation sequence. We will denote the validation sequence as \( \langle o_1, o_2, \ldots, o_k \rangle \), where \( o_i = (l_i, p_i) \), \( l_i \in \mathcal{L} \) and \( p_i \in \mathcal{P} \).

The schematic description of the validation process is presented in Figure 1. The following examples describe several applications in which validation can play an important role.

**Example 1** [Validation in Databases] \( D \) can be a set of records in a relational table that have to be classified into several categories by a domain expert. For example, a table can contain records of job candidates, where the human resources manager has to decide which candidates should be extended a job offer, which should be called for an additional interview, and which should not be considered for a position anymore. In this example, the labels of validation operators would correspond to various categories of job candidates, such as *job offer*, *additional interview*, and *reject*. The predicates would be various SQL queries issued by the domain expert on the candidate table, such as “Find the candidates who have excellent communication skills and who passed their previous job interviews with ratings of at least 5.”
Example 2 [Validation in Data Streams] $D$ can be a packet stream that goes through a router (or a firewall) of a certain Internet Service Provider (ISP). An example of a specific data stream processing task in this context could be: “monitor the amount of common traffic flowing through four routers, $R_1$, $R_2$, $R_3$, and $R_4$, in the ISP’s network over the last 10 minutes” [6]. The network analyst could use this task, for example, to monitor network health and find opportunities for load balancing [6]. As the result, the data streams from the above routers are then processed (validated) by a sequence of pipelined filters [6], which would correspond to validation operators in our framework.

As demonstrated by the examples above, the validation problem presented in this paper is general and can be applied in a variety of applications. However, since expert-driven validation is a very important part of many data mining applications, in order to illustrate the main concepts and theoretical results, we will use mostly data mining-related examples throughout this paper, including the one below.

Example 3 [Validation in Data Mining] Assume that $D$ is a set of association rules [5] about customer purchasing behavior that were discovered in a market basket analysis application. An example of such a rule would be “people who buy milk and yogurt, also buy bread”, or, more formally, milk & yogurt $\rightarrow$ bread. Assume that the set of labels $\mathcal{L}$ is defined as $\mathcal{L} = \{ \text{good, bad} \}$. In other words, the domain expert wants to label the association rules that she found to be of interest as “good” and the rules that she found of no interest (e.g., irrelevant or obvious rules) as “bad.” All the “good” rules would then comprise set $V_{\text{good}}$, and all the “bad” ones set $V_{\text{bad}}$. Since the number of discovered association rules can be very large in some applications, e.g., measured in hundreds of millions in some personalization or bioinformatics applications [3, 23], it may be very hard for the domain expert to validate all the rules. Therefore, some rules in $D$ are likely to remain unvalidated, and all such rules are placed into the “unvalidated” rule set $U$.

One way to specify validation operators for association rules would be to use various rule template specification languages [14, 22, 13, 3, 24]. A rule template takes a set of rules as an input and returns only those rules that satisfy
the template. For example, an expert may be interested to find the rules that make inferences about bread purchasing. This can be achieved by filtering the discovered association rules to return only the ones having attribute bread in their heads (consequents). Using the template specification language described in [3], this filter can be specified as the following validation predicate: “HEAD = bread”. Moreover, the expert may be interested in all such rules and accept them as “interesting” by assigning the label “good” to them. Formally, the validation operator in this case would be \( o = (\text{good}, \text{“HEAD = bread”}) \). For example, rule milk & yogurt \( \rightarrow \) bread would match the above template and, therefore, would be labeled as “good,” whereas rule bread \( \rightarrow \) butter would not match the above template and would remain unvalidated.

As another example, to filter the discovered rules that have milk (possibly among other attributes) in their bodies (antecedents), one can specify the following rule template: “BODY \( \supseteq \) milk”. Again, rule milk & yogurt \( \rightarrow \) bread would match this template, and rule bread \( \rightarrow \) butter would not.

The process of generating validation sequences in data mining applications is described in [2, 3]. In that work, it is argued that this is the job of the domain expert to drive the validation process since it occurs during the post-analysis stage of the knowledge discovery process when all the patterns are already discovered and these discoveries need to be validated by the domain expert.

As mentioned before, the general validation framework presented in this section generalizes the validation process that has been used in data mining literature [14, 13, 3, 24] and makes it applicable to various other validation tasks, such as validation of database records and data streams. However, once the domain expert selects a certain validation sequence, it may turn out that this sequence may not be the “most efficient” validation sequence since there can be “equivalent” sequences that perform the validation task more efficiently. Therefore, in this paper we focus on the performance optimization problem, i.e., how to select the most efficient sequence that is equivalent to the validation sequence initially specified by a domain expert. In other words, we address the problem of optimizing this sequence to be as efficient as possible, while retaining the exact validation behavior initially encoded by the domain expert.

The sequence optimization problem is relevant in a number of applications. Examples
of such applications include the data mining applications where data continuously changes over time, such as e-business customer profiling applications [3], and telecommunication applications, where there is a need to process continuous data streams. In such applications, the data that needs to be validated (i.e., the new discovered rules or the subsequent portion of the data stream) is not static – it is constantly changing over time, and therefore, needs to be periodically re-validated. However, it would be the waste of human resources to let the domain expert conduct this re-validation process all the time. A more efficient approach would be for the domain expert to validate the data (or a sample of data) only once and generate a validation sequence as a result. Then, the subsequent periodic validations can be performed automatically by the system using the validation sequence initially generated by the domain expert. Since the sequence initially generated by the domain expert may not be the most efficient, it may result in the waste of computing resources during subsequent validations. Therefore, it is important to optimize the initial validation sequence, the improved version of which could then be automatically applied for the further validations.

In the next section, we formally state the sequence optimization problem, including the definition of equivalence of validation sequences.

3. Sequence Optimization Problem

In this section, we formalize the sequence optimization problem, i.e., the selection of the most efficient validation sequence from the set of all the validation sequences that are equivalent to a specified sequence. However, before formulating the optimization problem, we first need to define the concepts of equivalence and the performance measure for validation sequences.

As mentioned earlier, as a result of the validation by sequence $s$, the input set $D$ is divided into $n+1$ pairwise disjoint sets $V_1, V_2, \ldots, V_n$, and $U$, where each $V_i$ represents the subset of $D$ that was labeled with $L_i$, and $U$ denotes the subset of $D$ that remains unlabeled after the validation process. We will denote $VI_s(D) := (V_1, V_2, \ldots, V_n)$ (validated input elements) and $UI_s(D) := U$ (unvalidated input elements).

3.1 Equivalence of Validation Sequences

Intuitively, we define two validation sequences to be equivalent if they always produce the same validation results for any data set to which they are applied. The formal definition of validation sequence equivalence is provided below.
Definition 3 (Equivalent Validation Sequences) Validation sequences $s$ and $s'$ are equivalent if and only if $VI_s(D) = VI_{s'}(D)$ for every input $D$. If $s$ and $s'$ are equivalent, we will denote it as $s \sim s'$. If $s$ and $s'$ are not equivalent, we will denote it as $s \not\sim s'$.

Based on the above definition, it is easy to see that the equivalence of two validation sequences can also be defined as stated in the following lemma.\(^1\)

Lemma 1 $s \sim s' \iff VI_s(\mathcal{E}) = VI_{s'}(\mathcal{E})$.

The equivalence of validation sequences can be viewed as a binary relation on the set of all possible validation sequences. Let's denote this relation $R_\sim$. Note, that $R_\sim$ is a true equivalence relation, since it is reflexive, symmetric, and transitive, as it immediately follows from Definition 3.

Lemma 2 Relation $R_\sim$ is a true equivalence relation.

Example 4 [Equivalent Validation Sequences] Consider the following two validation sequences (for validating association rules) that are expressed using the template specification language mentioned in Example 3. Assume that $\mathcal{L} = \{ \text{good}, \text{bad} \}$. Let $s = \langle (\text{good}, \text{"HEAD = Bread"}), (\text{good}, \text{"BODY $\supseteq$ Milk"}) \rangle$ and let $s' = \langle (\text{good}, \text{"BODY $\supseteq$ Milk"}), (\text{good}, \text{"HEAD = Bread"}) \rangle$. Both sequences have two validation operators and clearly $s \neq s'$ since $s'$ is a reverse of $s$. However, $s \sim s'$, since both $s$ and $s'$ will validate (i.e., will label with good) the same inputs, given any input dataset.

Now that we have defined the notion of validation sequence equivalence, we can formulate the validation sequence optimization problem more precisely, i.e.,

$$s^* = \arg\min_{s' \sim s} \text{cost}(s')$$

where $\text{cost}(s)$ is some performance measure for validation sequences.

This problem, as defined by (2), constitutes some variant of a scheduling problem. According to this problem, we need to rearrange (reschedule) operators in $s$ in order to minimize the cost function. We will explore this connection to scheduling further in Section 5.4.

In order to address the problem specified in (2), we should consider the following issues:

\[^1\]The proofs of the main theoretical results are presented in the appendix. Other proofs can be found in [1].
• How should the cost(s) function be defined?

• How do we find the sequences that are equivalent to s, i.e., what is the search space of our optimization problem?

• How to find the optimal sequence efficiently? That is, even if we know how to determine all possible equivalent sequences and how to calculate the cost function for each of them, there may be too many of these sequences, and therefore, the exhaustive search may not be practical for solving optimization problem (2).

Note that the problem defined by (2) assumes that the initial sequence is given and finds some better sequence equivalent to s. As was explained before, the issue of how the initial sequence s is generated lies beyond the scope of this paper and has been addressed elsewhere. For example, in [3] the initial sequence is generated by the domain expert for data mining applications during the post-analysis stage of the data mining process.

### 3.2 Defining the Cost of Validation Sequence

As mentioned in Section 2, this paper focuses on the problem of validation sequence efficiency. Therefore, the amount of time it takes to perform validation would be one natural way to define the cost of a validation sequence. However, note that the cost of validating a sequence may depend significantly on the input dataset to which the sequence is being applied. For example, on the one hand, validation sequence s might be able to validate the entire dataset $D_1$ with its first validation operator (i.e., if all elements of $D_1$ match the predicate of the first operator), therefore the subsequent validation operators would not even have to be invoked. On the other hand, the same validation sequence s might not be able to validate any elements of dataset $D_2$. That is, all validation operators would have to be invoked on all input elements of $D_2$.

Therefore, we define the validation sequence optimization problem in the context of a specific dataset $D$. That is, given expert-specified validation sequence s and input dataset D, which of the sequences that are equivalent to s would have the best performance on dataset $D$? This approach enables the validation process to be adaptive, i.e., if the characteristics of data change over time, the validation sequence can always be dynamically adjusted (optimized) based on its performance on the latest portion of data $D$. 


Ideally, we would like to use the running time as the performance measure for validation sequences. While we generally would be able to compute the running time of the expert-specified sequence \( s \) (i.e., during the initial validation process), it obviously would be difficult to estimate running time of other sequences theoretically. However, we can overcome this difficulty with certain simplifying assumptions. In particular, we will assume that it takes a certain fixed time for an arbitrary validation operator \( o = (l, p) \) to validate an arbitrary input element \( e \in D \), i.e., to check whether \( e \) satisfies predicate \( p \). In other words, we simply assume that all predicates are “similar” in their capabilities and that all individual predicate/input checks take about the same amount of time. Note, that this assumption is often used in similar situations; for example, [6] use this assumption to developing algorithms for optimizing the sequence of pipelined filters for data stream processing. Based on this assumption, we can define the cost function (given a validation sequence \( s \) and input \( D \)) to be the total number of predicate/input satisfaction checks performed by \( s \) to validate input \( D \). More specifically, assume that sequence \( s \) consists of \( k \) validation operators \( o_1, o_2, \ldots, o_k \) and each of the operators \( o_i \) validated \( n_i \) number of elements from input dataset \( D \). That is, operator \( o_1 \) checked all \(|D|\) inputs and validated \( n_1 \) of them. Subsequently, \( o_2 \) checked the remaining \(|D| - n_1\) inputs and validated \( n_2 \) of them, and so on. Then, the cost of validating \( D \) using sequence \( s \) can be defined as:

\[
\text{cost}(s, D) = |D| + (|D| - n_1) + (|D| - n_1 - n_2) + \ldots + (|D| - \sum_{j=1}^{k-1} n_j)
\]

\[
= k|D| - \sum_{i=1}^{k-1} (k - i) n_i
\]

(3)

Furthermore, let’s define \( \text{cost}_0(s, D) \) to be the worst possible cost scenario, when not a single input from \( D \) is validated by \( s \), i.e., all \( n_i = 0 \). Therefore, \( \text{cost}_0(s, D) = k|D| \). Then, we can define \( \text{benefit} \) function as follows:

\[
\text{benefit}(s, D) = \text{cost}_0(s, D) - \text{cost}(s, D)
\]

\[
= \sum_{i=1}^{k-1} (k - i)n_i
\]

(4)

Based on the above definitions of cost and benefit functions, our optimization problem is now formulated as:

\[
s^* = \arg \min_{s' \sim s} \text{cost}(s', D) = \arg \max_{s' \sim s} \text{benefit}(s', D)
\]

(5)
Now that we have defined the cost and benefit functions, we have to decide how we will be searching for equivalent sequences. An important question is: in what ways are we allowed to change sequence $s$, so that the newly obtained sequence $s'$ remains equivalent to $s$? Note, that the general sequence equivalence specified in Definition (3) constitutes a semantic concept, and, therefore, it is difficult to find a general solution to problem (5) over the space of all the equivalent sequences. This situation is common to a number of problems in computer science, for example, it is similar to the problem of finding equivalent schedules of transaction executions in databases for the purpose of concurrency control [7]. The equivalence of two schedules of transactions constitutes a semantic concept that is hard to verify formally. Therefore, for the purpose of analytical tractability, this semantic notion of equivalence was replaced with a simpler syntactic notion of serializability of a schedule of transactions [7].

In this paper, we follow a similar approach and replace the semantic notion of equivalence of validation sequences with a more restrictive but a simpler syntactic concept of equivalence between the permutations of some validation sequence. To illustrate this concept, consider Example 4 from Section 3.1. In that example, we had two equivalent sequences $s$ and $s'$, where $s'$ was a permutation of $s$. That is, $s'$ had exactly the same operators as $s$, but not necessarily in the same order. Therefore, given validation sequence $s$, we will be searching for the solution to our optimization problem among the permutations $s'$ of sequence $s$ such that $s \sim s'$.

The permutation-based approach allows us to keep our validation framework very general and not consider a particular structure of data and validation predicates since all we need to do in the permutation-based approach is to examine the positions of the predicates in the validation sequence. We would not be able to do this with the semantics-based approach to sequence equivalence, because it would require taking into account the complexities of the domain knowledge pertaining to data and predicates used in each specific application.

4. Permutation-Based Approach to Sequence Optimization

As explained before, given validation sequence $s$, we will be searching for the solution to our optimization problem among the permutations $s'$ of sequence $s$ such that $s \sim s'$. We would like to point out that the permutation-based approach has been used for optimizing sequences
of pipelined filters in the data stream processing applications [6]. However, [6] analyzes the optimization problem in the context where all filters are commutative, i.e., any two filters can be switched without any consequences to the correctness of the processing result (only the processing time may be affected). Such an approach is not applicable to our case since not all validation operators are commutative for our validation problem in general, as will be demonstrated below. Moreover, unlike [6], we deal with a general validation problem that goes beyond validation of data streams. Therefore, this problem needs a different permutation-based approach, and we present it below.

4.1 Permutations of Validation Sequences

Definition 4 (Sequence Permutation) Let $s$ and $s'$ be validation sequences, i.e., $s = <o_1, o_2, \ldots, o_k>$ and $s' = <o'_1, o'_2, \ldots, o'_k>$. Sequence $s'$ is called a permutation of sequence $s$ if and only if the sets $\{o_1, o_2, \ldots, o_k\}$ and $\{o'_1, o'_2, \ldots, o'_k\}$ are equal.

In other words, $s'$ is a permutation of $s$ if it contains exactly the same validation operators, but not necessarily in the same order.

Let $s = <o_1, o_2, \ldots, o_k>$ be a validation sequence and let $u$ be some validation operator. We will say that $s$ contains $u$ (and denote $u \in s$) if there exists $x \in \{1, \ldots, k\}$ such that $u = o_x$. In such case we will say that $s$ contains $u$ at a position $x$ and denote $pos_s(u) = x$.

Let $u$ and $v$ be validation operators contained in the validation sequence $s$, i.e., $u, v \in s$. Then we will say that $u$ precedes $v$ in sequence $s$ if $pos_s(u) < pos_s(v)$. We will denote it as $u \prec_s v$.

Definition 5 (Permutation Distance) Let $s$ be a validation sequence, i.e., let $s = <o_1, o_2, \ldots, o_k>$. Let $s' = <o'_1, o'_2, \ldots, o'_k>$ be some permutation of $s$. We will define the distance between $s$ and $s'$ as the number of inversions between $s$ and $s'$, i.e., the number of all distinct pairs of validation operators $u, v$, such that $u \prec_s v$ and $v \prec_s u$. More precisely,

$$\text{dist}(s, s') := |\{(u, v) : u \in s, v \in s, u \prec_s v, v \prec_s u\}|$$

Example 5 [Permutation Distance] Let $s = <o_1, o_2, o_3, o_4>$ and $s' = <o_3, o_2, o_4, o_1>$. Then the distance between $s$ and $s'$ is equal to 4, because there are 4 operator inversions in $s'$ with respect to $s$: $(o_3, o_1), (o_2, o_1), (o_4, o_1), (o_3, o_2)$.

Definition 6 (Simple Permutation) Let $s$ be a validation sequence, and let $s'$ be a permutation of $s$, such that $\text{dist}(s, s') = 1$. Then $s'$ is called a simple permutation of $s$. 

14
It follows from the above definitions that the validation sequence \( s' = < o'_1, o'_2, \ldots, o'_k > \) is a simple permutation of \( s = < o_1, o_2, \ldots, o_k > \) if and only if there exists \( i \in \{1, \ldots, k - 1\} \) such that \( o_i = o'_{i+1} \) and \( o_{i+1} = o'_i \), and for all \( j \) (\( j \neq i \) and \( j \neq i + 1 \)) \( o_j = o'_j \) holds true. Also note that, based on the above definitions, \( \text{dist}(s, s') \) represents the minimal number of simple permutations needed to obtain \( s' \) from \( s \).

**Definition 7 (Safe Permutation)** Let \( s \) be a validation sequence. Then \( s' \) is a safe permutation of \( s \) if and only if \( s' \) is a permutation of \( s \) and \( s \sim s' \).

The following lemma will be useful for proving some of the subsequent theoretical results.

**Lemma 3** If \( s' \) is a permutation of \( s \), then \( \text{UI}_s(D) = \text{UI}_{s'}(D) \) for every input dataset \( D \).

In the next two subsections we will prove several fundamental properties of validation sequence permutations.

### 4.2 Deriving Equivalence Criteria for Sequence Permutations

The following theorem states necessary and sufficient conditions for \( s \not\sim s' \), where \( s' \) is a permutation of \( s \).

**Theorem 4** Let \( s = < (l_1, p_1), \ldots, (l_k, p_k) > \) and \( s' = < (l'_1, p'_1), \ldots, (l'_k, p'_k) > \) be validation sequences, where \( s' \) is a permutation of \( s \). Then \( s \not\sim s' \) if and only if they contain a pair of validation operators \( u = (l_u, p_u) \) and \( v = (l_v, p_v) \) that satisfy all of the following conditions:

1. \( u \) precedes \( v \) in \( s \), but \( v \) precedes \( u \) in \( s' \), i.e., \( u \prec_s v \) and \( v \prec_{s'} u \);

2. \( u \) and \( v \) have different labels, that is, \( l_u \neq l_v \);

3. There exists an element \( e \in \mathcal{E} \) such that the following Boolean expression is true:

\[
p_u(e) \land p_v(e) \land \bigwedge_{i=1}^{x-1} \neg p_i(e) \land \bigwedge_{j=1}^{y-1} \neg p'_j(e)
\]

where \( x = \text{pos}_s(u) \) and \( y = \text{pos}_{s'}(v) \).

Intuitively, \( s \not\sim s' \) if and only if there exists a data element \( e \in \mathcal{E} \) such that \( e \) is validated differently by \( s \) and \( s' \), and the above theorem, the proof of which is presented in the Appendix, provides the necessary and sufficient conditions for this.
Example 6 [Non-equivalent Validation Sequences] Let \( s = < (\text{good}, \text{"HEAD = Bread"}), (\text{bad}, \text{"BODY \supseteq Milk"}) > \) and let \( s' = < (\text{bad}, \text{"BODY \supseteq Milk"}), (\text{good}, \text{"HEAD = Bread"}) > \). Both sequences have two validation operators and clearly \( s \neq s' \) since \( s' \) is a reverse of \( s \). It is easy to see that \( s \) and \( s' \) satisfy the first two conditions of Theorem 4. In order to see that the third condition of this theorem is also satisfied, consider rule \( \text{Milk} \rightarrow \text{Bread} \). Since this rule matches both of the predicates in \( s \) (and its permutation \( s' \)), sequence \( s \) would label this rule as \text{good} and \( s' \) would label it as \text{bad}. Hence, \( s \not\sim s' \).

The following corollary states necessary and sufficient conditions for \( s \sim s' \), where \( s' \) is a permutation of \( s \). It can be straightforwardly derived from Theorem 4 by taking the logical negation of the necessary and sufficient conditions for \( s \not\sim s' \).

**Corollary 5** Let \( s = < (l_1, p_1), \ldots, (l_k, p_k) > \) and \( s' = < (l'_1, p'_1), \ldots, (l'_k, \ldots, p'_k) > \) be validation sequences, where \( s' \) is a permutation of \( s \). Then \( s \sim s' \) if and only if every possible pair of validation operators \( u = (l_u, p_u) \) and \( v = (l_v, p_v) \) \( (i.e., u \in s, v \in s, u \neq v) \) must satisfy at least one of the following conditions:

1. Either \( u \) precedes \( v \) in both \( s \) and \( s' \), or \( v \) precedes \( u \) in both \( s \) and \( s' \);
2. \( u \) and \( v \) have the same label, that is, \( l_u = l_v \);
3. For all possible input elements \( e \in E \), the following Boolean expression is true:

   \[
   \neg(p_u(e) \land p_v(e)) \lor \bigvee_{i=1}^{x-1} p_i(e) \lor \bigvee_{j=1}^{y-1} p'_j(e) \tag{8}
   \]

   where \( x = \text{pos}_s(u) \) and \( y = \text{pos}_{s'}(v) \).

Finally, the following corollary states necessary and sufficient conditions for \( s \sim s' \), where \( s' \) is a simple permutation of \( s \). It follows immediately from Corollary 5 by restricting \( s' \) to be a simple permutation of \( s \).

**Corollary 6** Let \( s = < o_1, \ldots, o_k > \) be a validation sequence where \( o_i = (l_i, p_i) \), and let \( s' = < o'_1, \ldots, o'_k > \) be a simple permutation of \( s \). That is, \( (\exists x \in \{1, \ldots, k - 1\}) (o_x = o'_{x+1}) \land (o_{x+1} = o'_x) \), and also \( o_i = o'_i \) for all \( i \in \{1, \ldots, k\} \) such that \( i \neq x \) and \( i \neq x + 1 \). Then \( s \sim s' \) if and only if at least one of the following two conditions is satisfied:

16
1. \( o_x \) and \( o_{x+1} \) have the same label, that is, \( l_x = l_{x+1} \);

2. For all possible input elements \( e \in \mathcal{E} \), the following Boolean expression is true:

\[
\neg (p_x(e) \land p_{x+1}(e)) \lor \bigvee_{i=1}^{x-1} p_i(e) \tag{9}
\]

The practical implication of the first condition of the above corollary is clear: whenever we switch any two adjacent operators in a validation sequence (i.e., perform a simple permutation), the new sequence is equivalent to the initial one if the labels of the inverted operators are the same. It is somewhat more difficult to derive the precise practical implications of the second condition, and we will discuss this later in the paper.

### 4.3 “Connectedness” of Equivalent Sequence Permutations

In this section, we will prove several facts that will provide some understanding about the structure of the “space” of equivalent validation sequence permutations.

The following is a simple lemma that is useful for proving the subsequent results.

**Lemma 7** Let \( s = <o_1, o_2, \ldots, o_k> \) be a validation sequence. Let \( s' \) be some permutation of \( s \). Then, for every pair of validation operators \( o_i \) and \( o_j \) such that \( o_i \prec_s o_j \) (i.e., \( i < j \)), but \( o_j \prec_{s'} o_i \), there exists \( x \) such that \( i \leq x \leq j - 1 \) and \( o_{x+1} \prec_{s'} o_x \).

The following theorem constitutes the main result about the space of all equivalent permutations.

**Theorem 8** Let \( s = <o_1, o_2, \ldots, o_k> \) be a validation sequence. Let \( s' \) be some safe permutation of \( s \) such that \( \text{dist}(s, s') = d \), where \( d \geq 1 \). Then there exists a sequence \( s'' \) that is a safe simple permutation of \( s \), such that \( s' \sim s'' \) and \( \text{dist}(s', s'') = d - 1 \).

This theorem states that, if \( s \sim s' \), where \( s' \) is a permutation of \( s \) and \( \text{dist}(s, s') = d \), then there always exists an “intermediate” equivalent validation sequence \( s'' \) that is a simple permutation of \( s \), i.e., \( s'' \) equivalent to both \( s \) and \( s' \), \( \text{dist}(s, s'') = 1 \), \( \text{dist}(s', s'') = d - 1 \).

By applying the above theorem repeatedly, it is easy to demonstrate the “connectedness” of all equivalent validation sequences via their equivalent simple permutations, as stated by the following theorem.
Figure 2: Permutation graph of a validation sequence.

**Theorem 9** Let $s = < o_1, o_2, \ldots, o_k >$ be a validation sequence. Let $s'$ be some permutation of $s$ such that $\text{dist}(s, s') = d$, where $d \geq 1$. Then, $s \sim s'$ if and only if there exists $d + 1$ validation sequences $s_0, s_1, \ldots, s_d$, such that $s_0 = s$, $s_d = s'$, and $s_i$ is a safe simple permutation of $s_{i-1}$ for every $i = 1, \ldots, d$.

The above results give us a better understanding about equivalent permutations of a given validation sequence $s$. We can visualize the space of all permutations of $s$ by constructing a permutation graph, where each vertex corresponds to a different permutation of $s$. Furthermore, in this graph, two vertices will have an edge connecting them if one of them is a simple permutation of the other. An example of such a graph for a sequence consisting of four validation operators is presented in Figure 2.

Obviously, any two permutations are connected by a path (in fact, multiple paths) in this permutation graph. Theorem 9 states that if two permutations, say $s_1$ and $s_2$, are equivalent, there exists a minimal path from $s_1$ to $s_2$ that goes only through “intermediate” permutations that are equivalent to $s_1$ and $s_2$. Furthermore, not only such path exists, but it is also minimal, i.e., its length (i.e., number of edges comprising the path) is always
equal to the permutation distance between \( s_1 \) and \( s_2 \). Figure 2 illustrates this fact. To illustrate this, assume that \( s_1 := < o_1, o_2, o_3, o_4 > \), and let \( s_2 \) be a permutation of \( s_1 \) such that \( s_2 := < o_4, o_3, o_1, o_2 > \). Then the distance between \( s_1 \) and \( s_2 \) is 5, since there are 5 operator inversions in \( s_2 \) with respect to \( s_1 \), i.e., \((o_4, o_3), (o_4, o_2), (o_4, o_1), (o_3, o_2), (o_3, o_1)\). Figure 2 highlights all the paths of length 5 between \( s_1 \) and \( s_2 \).

Theorem 9 is very useful since it identifies a “search space” of equivalent validation sequence permutations, and we can take advantage of this in solving our sequence optimization problem. Let’s assume that we have an expert-specified validation sequence \( s \). Let’s also assume that we have a cost function defined for each permutation of \( s \) (we will address this issue in later sections). Then, we could search for the optimal permutation by traversing the permutation graph (such as depicted in Figure 2) by doing only safe simple permutations, starting from vertex \( s \). Theorem 9 guarantees that we will encounter the optimal permutation \( s^* \) along the way.

We could use various graph traversal techniques, such as depth-first search or breadth-first search. However, if the validation sequence \( s \) has \( k \) validation operators, then the number of possible permutations is \( k! \). When \( k \) is large, the exhaustive search techniques would not be scalable. On the other hand, obviously, the largest possible distance between two permutations is \((k-1)+(k-2)+\ldots+2+1 = k(k-1)/2\) (i.e., the largest possible number of inversions between two permutations – where every pair of operators is inverted). Therefore, according to Theorem 9, there exists a path between the expert-specified validation sequence \( s \) and the optimal sequence \( s^* \) that is not longer than \( k(k-1)/2 \). Consequently, based on the specific cost function, we would like to find a greedy algorithm that allows us to choose the right path and not have to traverse the whole graph.

One obstacle with this approach is that we must be able to determine whether a given simple permutation is safe. Corollary 6 gives us two conditions – at least one of them has to be met by a simple permutation in order for it to be safe. Assume \( s = < o_1, \ldots, o_k > \) be a validation sequence where \( o_i = (l_i, p_i) \), and let \( s' = < o'_1, \ldots, o'_k > \) be a simple permutation of \( s \). That is, \((\exists! x \in \{1, \ldots, k-1\})((o_x = o'_{x+1}) \land (o_{x+1} = o'_x)), \) and also \( o_x = o'_x \) for all \( i \in \{1, \ldots, k\} \) such that \( i \neq x \) and \( i \neq x + 1 \). As mentioned earlier, the first condition of Corollary 6, \( l_x = l_{x+1} \), is easily verifiable. The second condition, i.e., whether the following expression holds for all possible inputs \( e \in E \)

\[
\neg(p_x(e) \land p_{x+1}(e)) \lor \bigvee_{i=1}^{x-1} p_i(e)
\]
depends on the semantics of predicates \( p_i \) and in general may not be easily solvable analyti-
cally.

We will show later that even when we restrict the validation sequence optimization prob-
lem, it remains computationally complex. More specifically, we will show that even a special
case of the problem of finding optimal permutations of validation operators is NP-hard.

To study computational complexity of the validation sequence optimization problem and
to further understand the structure of the space of the equivalent validation sequence per-
mutations, we introduce in Section 5 the concepts of strong and very strong equivalence
of sequence permutations and study their properties. We later show that the optimization
problem stated above and even several of its special cases belong to the class of NP-hard
problems.

5. Computational Complexity of the Validation Sequence
Optimization Problem

In this section we consider two special classes of equivalent validation sequence permuta-
tions that are based on the concept of orthogonality of predicates. We will call them strongly
and very strongly equivalent permutations and will explore their various theoretical prop-
erties. Moreover, we will use the class of very strongly equivalent permutations to restrict
our sequence optimization problem and to show that this restricted optimization problem
is NP-hard. This implies that the general optimization problem is NP-hard as well. Fur-
thermore, we will use the less restrictive of the two classes – the class of strongly equivalent
validation sequence permutations – in creating heuristic algorithms for validation sequence
improvement.

Before defining strong and very strong equivalences, in the next section we introduce a
notion of orthogonality of validation predicates.

5.1 Predicate Orthogonality

In this section, we define the concept of predicate orthogonality and prove some of its basic
properties.

**Definition 8 (Predicate Orthogonality)** Two predicates \( p \) and \( q \) are orthogonal if their
conjunction is not satisfiable, i.e., if \( (\forall e \in E) \neg(p(e) \land q(e))) \). If \( p \) and \( q \) are orthogonal, we
will denote it \( p \perp q \). If \( p \) and \( q \) are not orthogonal, we will denote it \( p \not\perp q \).
Simply put, two predicates are orthogonal if they can never both match the same input element.

**Example 7** [Orthogonal Predicates] According to rule template specification language proposed in [3], rule templates “HEAD = bread” and “HEAD = milk” clearly would never match the same association rule. Hence, predicates “HEAD = bread” and “HEAD = milk” are orthogonal.

**Example 8** [Non-orthogonal Predicates] Rule templates “HEAD = bread” and “BODY \(\supseteq\) milk” would both match association rule milk \(\rightarrow\) bread. Hence, predicates “HEAD = bread” and “BODY \(\supseteq\) milk” are not orthogonal.

The following lemma follows immediately from the predicate orthogonality definition and the basic rules of logic.

**Lemma 10** \((\forall p \in \mathcal{P})(p \perp \neg p)\).

Predicate orthogonality can be viewed as a *binary relation* on the set of all predicates. We will denote this relation as \(R_\perp\). \(R_\perp\) has the properties specified in the following lemma. (These are the properties that one may intuitively expect an orthogonality relation to have, e.g., orthogonality relation on the set of straight lines on a two-dimensional plane has exactly the same properties.)

**Lemma 11** Binary relation \(R_\perp\) is symmetric, but neither reflexive nor transitive.

The following lemmas state the orthogonality conditions for predicate conjunctions and disjunctions.

**Lemma 12** Let \(p_1, \ldots, p_m\) and \(q\) be predicates. Let \(p := p_1 \land \ldots \land p_m\). If there exists \(i \in \{1, \ldots, m\}\) such that \(p_i \perp q\), then \(p \perp q\).

**Lemma 13** Let \(p_1, \ldots, p_m\) and \(q\) be predicates. Let \(p := p_1 \lor \ldots \lor p_m\). Then \(p \perp q\) if and only if \((\forall i \in \{1, \ldots, m\}) (p_i \perp q)\).

Once we defined predicate orthogonality and studied its properties, we are ready to define strong equivalence of permutations.
5.2 Strongly Equivalent Permutations

Definition 9 (Strongly Equivalent Permutation)

Let \( s = (l_1, p_1), \ldots, (l_k, p_k) \) and \( s' = (l'_1, p'_1), \ldots, (l'_k, p'_k) \) be validation sequences, where \( s' \) is a permutation of \( s \). \( s' \) is said to be a strongly equivalent permutation of \( s \) if and only if every possible pair of validation operators \( u = (l_u, p_u) \) and \( v = (l_v, p_v) \) (i.e., \( u \in s \), \( v \in s \), \( u \neq v \)) satisfies at least one of the following conditions:

1. Either \( u \) precedes \( v \) in both \( s \) and \( s' \), or \( v \) precedes \( u \) in both \( s \) and \( s' \);
2. \( u \) and \( v \) have the same label, that is, \( l_u = l_v \);
3. \( p_u \) and \( p_v \) are orthogonal, i.e., \( p_u \perp p_v \).

If \( s' \) is a strongly equivalent permutation of \( s \), we will denote it as \( s \approx s' \).

Note, that the above definition mirrors the necessary and sufficient conditions for sequence equivalence (Corollary 5), except that the third condition is strengthened here.\(^2\) The following lemma, which follows straightforwardly from the above definition and Corollary 5, indicates the relationship between strongly equivalent and equivalent permutation sequences.

Lemma 14 \( s \approx s' \implies s \sim s' \)

Example 9 [Strongly Equivalent Permutations] Consider validation sequences from Example 4, i.e., \( s = (\text{good}, "\text{HEAD} = \text{Bread}"), (\text{good}, "\text{BODY} \supseteq \text{Milk}")) \) and its permutation \( s' = (\text{good}, "\text{BODY} \supseteq \text{Milk}"), (\text{good}, "\text{HEAD} = \text{Bread}")) \). According to Definition 9, we have that \( s \approx s' \), since the permuted validation operators have the same label, i.e., “good.”

The strong equivalence of validation sequence permutations can be viewed as a binary relation on the set of all possible permutations of a given validation sequence. Let’s denote this relation \( R_\approx \). Note, that \( R_\approx \) is a true equivalence relation, as the following lemma shows.

Lemma 15 Relation \( R_\approx \) is a true equivalence relation, i.e., \( R_\approx \) is reflexive, symmetric, and transitive.

\(^2\)Hence the term “strong equivalence.”
Also, for strongly equivalent sequence permutations there exist necessary and sufficient conditions as well as “connectedness” results that are very similar to the corresponding results for equivalent permutations (i.e., Corollary 6 and Theorem 9), as stated below.

**Theorem 16** Let \( s = < o_1, \ldots, o_k > \) be a validation sequence where \( o_i = (l_i, p_i) \), and let \( s' = < o'_1, \ldots, o'_k > \) be a simple permutation of \( s \). That is, \( (\exists! \ x \in \{1, \ldots, k-1\}) \left((o_x = o'_{x+1}) \land (o_{x+1} = o'_x)\right) \), and also \( o_i = o'_i \) for all \( i \in \{1, \ldots, k\} \) such that \( i \neq x \) and \( i \neq x + 1 \).

Then \( s \approx s' \) if and only if at least one of the following two conditions is satisfied:

1. \( o_x \) and \( o_{x+1} \) have the same label, that is, \( l_x = l_{x+1} \);
2. \( p_x \bot p_{x+1} \).

**Theorem 17** Let \( s = < o_1, o_2, \ldots, o_k > \) be a validation sequence. Let \( s' \) be some permutation of \( s \) such that \( \text{dist}(s, s') = d \geq 1 \). Then, \( s \approx s' \) if and only if there exists \( d + 1 \) validation sequences \( s_0, s_1, \ldots, s_d \), such that \( s_0 = s, s_d = s' \), and \( s_i \) is a strongly equivalent simple permutation of \( s_{i-1} \) for every \( i = 1, \ldots, d \).

As mentioned earlier, we will use strongly equivalent sequence permutations in developing the heuristic algorithms for validation sequence improvement, as described in Section 6.

Now that we have defined the notion of the strong equivalence for validation sequence permutations, we can formulate the following special case of our validation sequence optimization problem (i.e., a restriction of the search space to very strongly equivalent permutations):

\[
\begin{align*}
\text{arg min}_{s' \approx s} \text{cost}(s', D) &= \text{arg max}_{s' \approx s} \text{benefit}(s', D) \\
&= \text{arg max}_{s' \approx s} \sum_{i=1}^{k-1} (k-i)n'_i
\end{align*}
\]

We know the number of data points \( n_i \) from \( D \) validated by each validation operator \( o_i \) in the initial sequence \( s \). However, the corresponding values \( n'_i \) for validation operators \( o'_i \) in \( s' \) are different from the \( n_i \) for strongly equivalent sequences \( s' \) in (10) in general. Since we do not know the \( n'_i \) values, this makes it difficult to solve the problem (10).

In the next section, we will consider a more restricted version of strong equivalence – very strong equivalence – that guarantees that the set of values \( \{n'_i\} \) is always the same.
as \( \{u_i\} \) for very strongly equivalent sequence permutations. This observation will be useful for determining the computational complexity of the optimization problem for that class of sequences and, consequently, for optimization problems (10) and (5).

### 5.3 Very Strongly Equivalent Permutations

In the previous subsection we introduced a subclass of equivalent permutations called strongly equivalent permutations. Let’s restrict the class of strongly equivalent permutations even further and introduce the class of *very strongly equivalent permutations*. We will use this new class to prove NP-hardness of the restricted optimization problem and, consequently, NP-hardness of the general optimization problem (5).

**Definition 10 (Very Strongly Equivalent Permutation)**

Let \( s = \langle (l_1, p_1), \ldots, (l_k, p_k) \rangle \) and \( s' = \langle (l'_1, p'_1), \ldots, (l'_k, p'_k) \rangle \) be validation sequences, where \( s' \) is a permutation of \( s \). \( s' \) is said to be a *very strongly equivalent permutation* of \( s \) if and only if every possible pair of validation operators \( u = (l_u, p_u) \) and \( v = (l_v, p_v) \) (i.e., \( u \in s, v \in s', u \neq v \)) satisfies at least one of the following conditions:

1. Either \( u \) precedes \( v \) in both \( s \) and \( s' \), or \( v \) precedes \( u \) in both \( s \) and \( s' \);
2. \( p_u \perp p_v \).

If \( s' \) is a very strongly equivalent permutation of \( s \), we will denote it as \( s \cong s' \).

The above definition essentially mirrors the definition of strongly equivalent permutations (Definition 9), except the second condition \( (l_u = l_v) \) is omitted here.

**Example 10 [Very Strongly Equivalent Permutations]** Consider validation sequence \( s = \langle \text{good, "HEAD = bread"}, \text{good, "HEAD = milk"} \rangle \) and its permutation \( s' = \langle \text{good, "HEAD = milk"}, \text{good, "HEAD = bread"} \rangle \). The condition \( s \cong s' \) holds because the predicates of permuted validation operators are orthogonal according to Example 7.

**Example 11 [Not Very Strongly Equivalent Permutations]** Consider validation sequence \( s = \langle \text{good, "HEAD = bread"}, \text{good, "BODY \supseteq milk"} \rangle \) and its permutation \( s' = \langle \text{good, "BODY \supseteq Milk"}, \text{good, "HEAD = Bread"} \rangle \). We have that \( s \not\cong s' \), because the predicates of permuted operators are not orthogonal, according to Example 8.
The next lemma follows immediately from the definitions.

**Lemma 18** \( s \cong s' \implies s \approx s' \)

One could also prove that the relation of very strong equivalence is a true equivalence relation in the same manner as was done for strongly equivalent validation sequences in Section 4.2. The “connectedness” results for very strongly equivalent permutations can also be obtained using the same techniques as were used to prove the “connectedness” for equivalent validation sequences in Section 4.3.

Let \( s' \) be a very strongly equivalent permutation of \( s \). Let’s consider an arbitrary operator \( o_i \in s \), i.e., \( \text{pos}_s(o_i) = i \). Also, let’s assume that in the permuted sequence \( s' \), \( o_i \) would be at some position \( j \), i.e., \( o_i = o'_j \) or \( \text{pos}_{s'}(o_i) = j \). Finally, let’s also assume that \( o_i \) validated \( n_i \) data points from dataset \( D \). How many data points will \( o_i \) validate in the permuted sequence \( s' \) being at a position \( j \)? The answer is, it will validate exactly \( n_i \) points again, regardless of position \( j \) it was placed in the permuted sequence \( s' \), as demonstrated below.

**Lemma 19** Let \( s = \langle o_1, \ldots, o_k \rangle \) be a validation sequence and let \( n_1, \ldots, n_k \) be the numbers of data points validated by each of the validation operators in \( s \), given some dataset \( D \). Let \( s' = \langle o'_1, \ldots, o'_k \rangle \) be a very strongly equivalent permutation of \( s \) (i.e., \( s \cong s' \)) and let \( n'_1, \ldots, n'_k \) be the numbers of data points validated by each of the validation operators in \( s' \), given the same dataset \( D \). Then for every \( i \in \{1, \ldots, k\} \): \( n_i = n'_j \) where \( j = \text{pos}_{s'}(o_i) \).

Let’s further restrict the optimization problem (10) to its special case:

\[
\text{Let } s^* = \arg \max_{s' \cong s} \text{benefit}(s', D) \tag{11}
\]

In the next section, we will show that this problem is NP-hard.

### 5.4 NP-Hardness of the Optimization Problem for Very Strongly Equivalent Permutations

We will show NP-hardness of the above optimization problem by reducing a known NP-hard task sequencing problem to it.

Given validation sequence \( s = \langle o_1, \ldots, o_k \rangle \) and some permutation \( s' \), Definition (10) specifies several conditions that must be satisfied by every pair of validation operators \( u \in s \) and \( v \in s \), so that \( s \cong s' \). We will show that these conditions are equivalent to specifying a certain partial order on the set of validation operators in \( s \).
Constructing a precedence graph  More specifically, let $G_s = (V, E)$ be a directed acyclic graph with $k$ vertices, where each vertex $i \in V$ is associated with a different validation operator $o_i$ ($i \in \{1, \ldots, k\}$). Furthermore, the set $E$ of edges is defined as follows. For every pair of validation operators $o_i = (l_i, p_i)$ and $o_j = (l_j, p_j)$ such that $o_i \prec_s o_j$ (i.e., $i < j$), add an edge from vertex $i$ to vertex $j$ to set $E$ if $p_i \not\perp p_j$. We will call this graph a precedence graph of sequence $s$.

Note, that if precedence graph $G_s$ has an edge from $i$ to $j$, then any permutation $s'$ that is very strongly equivalent to $s$ must have $o_i \prec_{s'} o_j$. If that were not the case, i.e., if there existed a very strongly equivalent permutation $s'$ such that $o_j \prec_{s'} o_i$, then we would derive a contradiction, since validation operators $o_i$ and $o_j$ would not satisfy either of the two conditions from Definition (10) and it would imply that $s \not\equiv s'$.

It is easy to see that $G_s$ represents a partial order over the set of validation operators $o_1, \ldots, o_k$. As we showed above, those permutations of $s$ that satisfy this partial order are very strongly equivalent to $s$, and the ones that do not satisfy this order are not very strongly equivalent to $s$. Also, since we have not placed any restrictions on what kind of predicates can be used in validation operators, the resulting precedence graph in general can represent any possible partial order.

Therefore, we have transformed the restricted validation sequence optimization problem

$$s^* = \arg\max_{s' \succeq s} \sum_{i=1}^{k-1} (k - i)n_i'$$

into the following validation operator scheduling problem:

$$s^* = \arg\max_{s' \succeq s} \max_{s' \succeq s} \sum_{i=1}^{k-1} (k - i)n_i'$$

(12)

where $n_i' = n_x$, if $pos_s(o_x) = i$ (according to Lemma 19). In other words, the problem is to find a “scheduling” of operators $o_1, \ldots, o_k$ such that it obeys the precedence graph $G_s$ and the corresponding permutation $\{n_i'\}$ of $\{n_i\}$ maximizes the benefit function.

Let’s assume that we can efficiently compute whether two predicates are orthogonal. Therefore, let’s assume that, given validation sequence $s$, we can efficiently construct precedence graph $G_s$. Then, we will show that the above scheduling problem (12) is NP-hard, by showing that solving it is equivalent to solving a known NP-hard problem, described below.
Task Sequencing to Minimize Weighted Completion Time  The following problem is often referred to as the problem of “Task Sequencing on a Single Processor to Minimize Weighted Completion Time” [11].

Assume, that a set of tasks $T$ has to be sequenced for processing by a single machine. The sequencing of the tasks must obey the precedence constraints imposed by a given directed acyclic graph $G = (V,E)$, where each vertex $v \in V$ is associated with a different task (therefore, $|T| = |V|$). In other words, $G$ imposes a partial order on $T$. Task $t' \in T$ must precede task $t'' \in T$ if there is a directed path from $t'$ to $t''$ in $G$.

Furthermore, each task $t$ is assigned a processing time $p(t) \in Z^+$ and a weight $w(t) \in Z$. Given a specific sequencing of $T$, e.g., $s = < t_1, \ldots, t_k >$, the completion time of each task $t_i$ is denoted as $C(t_i)$ and can be calculated as

$$C(t_i) = \sum_{j=1}^{i} p(t_j)$$

where we assume that the processing of the first task begins immediately (i.e., at time 0) and there is no idle time between consecutive jobs.

The objective of the sequencing problem is to find the feasible sequence (i.e., that obeys the partial order imposed by $G$) $s = < t_1, \ldots, t_k > (t_i \in T)$ that minimizes the weighted total completion time $WTCT(s)$, defined as a weighted sum of individual completion times, i.e.,

$$WTCT(s) = \sum_{i=1}^{k} w(t_i) C(t_i)$$

Lawler [15] showed that the above problem is NP-hard. Furthermore, it was also shown that the above problem remains NP-hard even when all $w(t) = 1$.

Assuming $w(t) = 1$ for all $t \in T$ and using the definition of $C(t)$ from Equation (13), the weighted total completion time of sequence $s = < t_1, \ldots, t_k >$ can be expressed as

$$WTCT(s) = \sum_{i=1}^{k} C(t_i) = \sum_{i=1}^{k} \sum_{j=1}^{i} p(t_j) = \sum_{i=1}^{k} (k + 1 - i) p(t_i)$$

Equivalence of the Two Problems  As indicated above, the problem of finding the task sequence that obeys the specified partial order and minimizes the weighted total completion time is NP-hard. We will show that the problem of finding the task sequence that obeys the specified partial order and maximizes the weighted total completion time is also NP-hard.
Let \( G = (V, E) \) be an acyclic directed graph representing the partial order to be imposed on tasks \( T \). Then we will define a “reverse” graph \( G' = (V', E') \) as follows. Let \( V' = V \) and let \( E' \) contain the same edges as \( E \), only each edge should point in the reverse direction. That is, \( E' := \{(u, v) : (v, u) \in E\} \).

As indicated in the following lemma, it can be shown that \( s = \langle t_1, \ldots, t_k \rangle \) minimizes \( WTCT \) with respect to partial order \( G \) if and only if \( s' = \langle t_k, \ldots, t_1 \rangle \) (i.e., \( s' \) is the reversed sequence \( s \)) maximizes \( WTCT \) with respect to partial order \( G' \).

**Lemma 20** \( s = \langle t_1, \ldots, t_k \rangle \) minimizes \( WTCT \) with respect to partial order \( G \) \( \iff \) \( s' = \langle t_k, \ldots, t_1 \rangle \) maximizes \( WTCT \) with respect to partial order \( G' \).

The above lemma indicates that solving the problem of task sequencing to minimize weighted completion time subject to partial order constraints is equivalent to solving the problem of task sequencing to maximize weighted completion time subject to partial order constraints. Since the former problem has been shown to be NP-hard [15], consequently the latter problem is NP-hard as well. In addition, the latter problem is equivalent to our restricted validation sequence optimization (i.e., benefit maximization) problem (Equation 12), since in both cases we are searching for the sequence that satisfies the given partial order and maximizes essentially the same function.\(^3\) Hence, our restricted optimization problem is NP-hard as well. The following theorem summarizes the above results.

**Theorem 21** Optimization problem defined in (11) is NP-hard.

The NP-hardness of the less restrictive optimization problem follows immediately from Theorem 21 and Lemmas 14 and 18.

**Corollary 22** Optimization problems defined in (5) and (10) are NP-hard.

**Validation as Scheduling** Besides demonstrating that the validation sequence optimization process is inherently computationally intractable (i.e., NP-hard), the above analysis also provides a valuable insight that the optimization process presented in this paper can be viewed as a certain type of scheduling. To illustrate this point further, consider the following analogy to be used subsequently for drawing connections to the scheduling problem.

\(^3\)Actually, the functions in two problems differ, but only by a constant that does not depend on a particular sequencing and, therefore, does not affect the solution.
Consider a factory that manufactures items of a certain kind. After the items are manufactured, they all have to be sorted according to their quality, e.g., excellent, good, medium, or bad (in general, the number of degrees of quality is arbitrary), therefore, they are placed on a moving conveyor belt for the final inspection. A group of inspection robots is standing along the side of this conveyor belt, and each robot inspects each passing item for a particular predefined property. For example, robot $R_1$ only knows that all the “defective” items should be labeled as bad, robot $R_2$ only knows that all “small” and “green” items should be labeled as good, robot $R_3$ only knows that all “big” and “red” items should be labeled as medium, and so on. If the item matches the inspection criteria for a robot, it is removed from the conveyor belt by that robot and is placed into an appropriately labeled container. If the item does not match the inspection criteria, the item stays on the conveyor belt and moves to the next inspector.

Moreover, assume that an inspection of a single item by a single robot costs a factory a certain amount $c$ (i.e., electricity, robot wear and tear, etc.). Therefore, the problem for the factory is to arrange the robots alongside the conveyor belt so as to minimize the total inspection costs. One way to solve this problem is to put the robots that “capture” more items at the beginning of the conveyor, if possible. To see this, consider the following example.

**Example 12** [Validation as Scheduling] Suppose, the above-mentioned three robots are initially arranged in a sequence: $R_1, R_2, R_3$. Furthermore, suppose, that after the first 1000 inspections, $R_1$ labeled 20 (out of 1000 items), $R_2$ labeled 20 (out of the remaining 980 items), and $R_3$ labeled 400 (out of the remaining 960 items). The total cost in this case is 2940 inspections.

However, if the sequence had been $R_1, R_3, R_2$, then $R_1$ again would have labeled 20 (out of 1000 items), $R_3$ would have labeled 400 (out of remaining 980), and $R_2$ would have labeled 20 (out of remaining 580). The total cost in this case would be 2560 inspections – a significant improvement over the previous sequence.

With respect to the validation optimization, obviously, the items on the conveyor belt represent the data points to be validated, the sequence of robots represents the sequence of validation operators, and the total inspection cost represents our validation cost function. Moreover, various rearrangements of robots along the conveyor belt represent various permutations of validation operators. Notice, that not all robot rearrangements are possible.
In the above example we could switch robots $R_2$ and $R_3$, because they represent orthogonal validation operators, i.e., no item can be both “small/green” and “big/red” at the same time. However, we could not have switched $R_1$ and $R_2$, because an item can be “small/green” and “defective” at the same time. Such item would be labeled differently than in the original sequence, thus producing an incorrect outcome.

As follows from this analogy, the validation problem presented in this paper can be formulated as a scheduling problem: given a set of validation operators \( \{o_1, o_2, \ldots, o_k\} \) schedule them in a sequence $s'$ in such as way that (a) $s'$ is equivalent to the original sequence $s$ specified exogenously (e.g., by the domain expert), and (b) its cost on some representative data set $D$, $\text{cost}(s', D)$, is minimal among all the equivalent sequences.

Although formulated as a scheduling problem, this problem constitutes a non-traditional scheduling problem for the following reasons. First, it depends on a concept of equivalence between two sequences that is defined in this paper differently from the previously proposed concepts in the scheduling theory. Second, the cost structure, as defined by equations (3) and (4), is such that the cost of each validation operator $o_i$ contributing towards the total validation cost not only depends highly on the position of $o_i$ in the validation sequence, but also on its relative position with respect to other validation operators. To the best of our knowledge, this also constitutes a non-standard assumption in scheduling theory.

For these reasons, we could not apply standard methods from the scheduling theory to solve the validation sequence optimization problem. Instead, we have utilized the theoretical results derived in this paper in the heuristic algorithms presented in the next section.

6. Greedy Heuristic for Validation Sequence Improvement

In the previous section we showed that, given validation sequence $s$, the problem of finding the optimal sequence among all the sequences that are very strongly equivalent to $s$ is NP-hard. Note, that this problem is already NP-hard without even taking into account the computation of the precedence graph $G_s$. In the case of the task scheduling problem, described in Section 5.4, the precedence graph is given (i.e., it is part of the input). However, in our problem, we have to calculate the precedence graph ourselves. In other words, we have to be able to calculate which pairs of operators must preserve their precedence in the permuted sequence, based on the orthogonality of their predicates.
Note, that the precedence graph calculation depends on the class of predicates used in validation operators. If the predicates are complex, it may be very difficult (or impossible) to show whether two given predicates are orthogonal or not. Since the problem is NP-hard even when the graph is already given, in this section we will present two general (i.e., independent of the class of predicates used) heuristic-based approaches to improving the validation sequence when a precedence graph \( G_s \) is given as an input.

### 6.1 Precedence Graph for Strongly Equivalent Permutations

For the purpose of proving the NP-hardness of the optimization problem, we showed earlier how to construct the precedence graph based on *very strong equivalence* constraints. For our heuristic approaches, we will construct the precedence graph based on less restrictive equivalence – *strong equivalence* – constraints. In this way, our heuristics will have a much larger search space to work with and, therefore, may generate permutations with better performance improvements.

Given validation sequence \( s = < o_1, \ldots, o_k > \) and some its permutation \( s' \), Definition 9 specifies several conditions that must be satisfied by every pair of validation operators \( u \in s \) and \( v \in s \) so that \( s \approx s' \). We will show that these conditions are equivalent to specifying a certain partial order on the set of validation operators in \( s \).

More specifically, let \( G_s = (V, E) \) be a directed acyclic graph with \( k \) vertices, where each vertex \( i \in V \) is associated with a different validation operator \( o_i \) \((i \in \{1, \ldots, k\})\). Furthermore, the set \( E \) of edges is defined as follows. For every pair of validation operators \( o_i = (l_i, p_i) \) and \( o_j = (l_j, p_j) \) such that \( o_i \prec_s o_j \) (i.e., \( i < j \)), add an edge from vertex \( i \) to vertex \( j \) to set \( E \) if both \( l_i \neq l_j \) and \( p_i \not\perp p_j \).

Note, that if precedence graph \( G_s \) has an edge from \( i \) to \( j \), then any permutation \( s' \) that is strongly equivalent to \( s \) must have \( o_i \prec_{s'} o_j \). If that were not the case, i.e., if there existed a strongly equivalent permutation \( s' \) such that \( o_j \prec_{s'} o_i \), then we would derive a contradiction, since validation operators \( o_i \) and \( o_j \) would not satisfy all three conditions from Definition 9 and it would imply that \( s \not\approx s' \).
6.2 Sequence Improvement using a Simple Permutation

Let $s = < o_1, \ldots, o_k >$ be a validation sequence. Also, let $G$ be a precedence graph based on sequence $s$. As mentioned earlier, the cost of $s$ given specific input data $D$ is:

$$\text{cost}(s, D) = k |D| - \sum_{i=1}^{k-1} (k - i) n_i$$

where $n_i$ is the number of input data points from $D$ validated (labeled) by validation operator $o_i$.

Also, let $s' = < o'_1, \ldots, o'_k >$ be a simple permutation of $s$. That is, $(\exists! x \in \{1, \ldots, k - 1\}) ((o_x = o'_{x+1}) \land (o_{x+1} = o'_x))$, and also $o_i = o'_i$ for all $i \in \{1, \ldots, k\}$ such that $i \neq x$ and $i \neq x + 1$. Also, let’s assume that there is no precedence constraint between operators $o_x$ and $o_{x+1}$, i.e., there is no edge from $o_x$ to $o_{x+1}$ in $G$. Therefore, $s \approx s'$. Consequently, $s \sim s'$ and therefore $s'$ will produce the same validation results as $s$. What is the cost of $s'$? To be able to answer this, we have to estimate the numbers $n'_i$, i.e., the number of data points from dataset $D$ that each validation operator $o'_i$ (from permuted sequence) would validate.

First, it is clear that $n_i = n'_i$ for all $i < x$, since only the operators $o_x$ and $o_{x+1}$ are permuted. That is, first $x - 1$ operators in both sequences $s$ and $s'$ are the same and will produce the same validation results.

It is also easy to see that $n_i = n'_i$ for all $i > x + 1$. This is the case, because the set of first $x + 1$ validation operators is the same in both sequences (not necessarily in the same order). Obviously, the exact same subset of input dataset $D$ would remain unvalidated after $x + 1$ operators in both.

In addition, $o_i = o'_i$ for $i > x + 1$. Therefore, $n_i = n'_i$ for all $i > x + 1$.

We still need to estimate $n'_x$ and $n'_{x+1}$. Let’s consider operators $o_x = (l_x, p_x)$ and $o_{x+1} = (l_{x+1}, p_{x+1})$. We know that $o'_x = o_{x+1}$ and $o'_{x+1} = o_x$. Since $s \approx s'$ and $s'$ is a simple permutation of $s$, according to Lemma 16 we have one of the following two possibilities:

- $p_x \perp p_{x+1}$. This means that validation operators $o_x$ and $o_{x+1}$ can never both match the same input data point. Therefore, it does not matter whether $o_x$ precedes $o_{x+1}$ (as in sequence $s$) or $o_{x+1}$ precedes $o_x$ (as in sequence $s'$), they will still validate the same exact data points as before. Hence, $n'_x = n_{x+1}$ and $n'_{x+1} = n_x$.

- $l_x = l_{x+1}$. Since $o_{x+1}$ will precede $o_x$ in sequence $s'$, obviously, it will be able to validate at least as many data points in $s'$ as in $s$, therefore $n'_x \geq n_{x+1}$. As mentioned above, the

\[\text{For more precise reasoning, consider the two sequences of length } x + 1 \text{ and see Lemma 3.}\]
set of first \( x + 1 \) validation operators is the same in both sequences (not necessarily in the same order) and the exact same subset of input dataset \( D \) would remain unvalidated after \( x + 1 \) operators in both. Therefore, \( \sum_{i=1}^{x+1} n_i = \sum_{i=1}^{x+1} n'_i \). However, since \( n_i = n'_i \) for \( (i < x) \), we have that \( n_x + n_{x+1} = n'_x + n'_{x+1} \). Furthermore, since \( n'_x \geq n_{x+1} \) (as we have just shown), we have that \( n'_{x+1} \leq n_x \).

Therefore, in both cases above it is true that \( n_x + n_{x+1} = n'_x + n'_{x+1} \) and \( n'_{x+1} \leq n_x \). Now, let’s estimate how much different is the cost of sequence \( s \) from the cost of sequence \( s' \), when \( s \approx s' \) and \( s' \) is a simple permutation of \( s \).

In general, let’s denote the improvement of sequence \( s' \) over sequence \( s \) as \( \Delta_{s \rightarrow s'} \) and define it as follows:

\[
\Delta_{s \rightarrow s'} := \text{cost}(s, D) - \text{cost}(s', D)
\] (16)

In other words, \( \Delta_{s \rightarrow s'} \) specifies how much more efficient \( s' \) is with respect to \( s \). Based on definitions of \textit{cost} and \textit{benefit} functions (Equations 3 and 4), it is obvious that:

\[
\Delta_{s \rightarrow s'} = \text{benefit}(s', D) - \text{benefit}(s, D)
\] (17)

In the case where \( s' \) is a simple permutation of \( s \), we get (by applying the above analysis and also by plugging in the definition of the \textit{benefit} function from Equation 4):

\[
\Delta_{s \rightarrow s'} = \text{benefit}(s', D) - \text{benefit}(s, D)
\]

\[
= \sum_{i=1}^{k-1} (k - i)(n'_i - n_i)
\]

\[
= (k - x)(n'_x - n_x) + (k - x - 1)(n'_{x+1} - n_{x+1})
\]

\[
= (k - x)(n'_x + n'_{x+1} - n_x - n_{x+1}) + (n_{x+1} - n'_{x+1})
\]

\[
= n_{x+1} - n'_{x+1}
\]

\[
\geq n_{x+1} - n_x
\] (18)

where in the last equation we have an equality in the case when \( p_x \) and \( p_{x+1} \) are orthogonal, as demonstrated in Lemma 19.

Therefore, we have that whenever we perform a simple permutation that is permissible (i.e., allowed by the precedence graph), we are guaranteed to decrease the cost (or increase the benefit) of the validation sequence by at least \( n_{x+1} - n_x \). Based on this simple idea, in the
next section we propose a greedy heuristic-based method for reducing the cost of validation sequences.

6.3 Greedy Heuristic Based on Simple Permutations

Our heuristic algorithm for improving validation sequences in part relies on the ability to calculate predicate orthogonality. Assume $s = <o_1, \ldots, o_k>$ is an expert-specified validation sequence that was used to validate dataset $D$, and let $n_i$ be the number of elements validated by operator $o_i$, where $i \in \{1, \ldots, k\}$. We will construct an improved validation sequence $s' = <o'_1, \ldots, o'_k>$ where $o'_i = (l'_i, p'_i)$ as a permutation of $s$. In the beginning, let $o'_i := o_i$ and $n'_i = n_i$ for each $i$. Then, the construction of $s'$ can be described as the following iterative process:

1. $\text{done} := \text{False}; \Delta_{\text{Total}} := 0$
2. while $\neg \text{done}$
   3. $S := \{ 1 \leq i \leq k - 1 \mid l_i = l_{i+1} \lor p_i \perp p_{i+1} \}$
   4. if $S = \emptyset$ then $\text{done} := \text{True}$
   5. else $x := \text{arg max}_{i \in S}(n'_{i+1} - n'_i)$
   6. $\Delta_x := n'_{x+1} - n'_x$
   7. if $\Delta_x \leq 0$ then $\text{done} := \text{True}$
   8. else $\Delta_{\text{Total}} := \Delta_{\text{Total}} + \Delta_x$
      swap $o'_x$ and $o'_{x+1}$ in the validation sequence $s'$
      swap values of $n'_x$ and $n'_{x+1}$
   11. end if
12. end if
13. end while

The intuition behind this algorithm is the following. In every iteration of the algorithm (lines 3-12), given the current sequence $s'$, in a greedy fashion we choose to make a simple permutation of $s'$ that remains equivalent to $s'$. That is, we consider only those pairs of adjacent operators $o'_i$ and $o'_{i+1}$ that either have identical labels or their predicates are orthogonal (line 3). Based on Definition 9, by swapping these two operators we would produce an equivalent permutation. Furthermore, in each iteration, among all available equivalent simple permutations we choose the one that maximizes the improvement of the sequence performance (line 5). The simple permutations (lines 9-10) continue in this greedy
fashion until there is no incremental improvement to the sequence performance (determined at line 7).

While the greedy heuristic is not guaranteed to give us an optimal solution to the sequence optimization problem, it can be shown that

$$\Delta_{s\to s'} \geq \Delta_{\text{Total}}$$

(19)

where $\Delta_{\text{Total}}$ is calculated by the above heuristic algorithm. More precisely,

$$\Delta_{\text{Total}} = \sum_{i: (o_i \prec_s o_j) \land (o_j \prec_s o_i)} (n_j - n_i)$$

(20)

To show that $\Delta_{s\to s'} \geq \Delta_{\text{Total}}$, one can straightforwardly extend the argument from Section 6.2 which showed that $\Delta_{s\to s'} \geq n_{x+1} - n_x$ for simple strongly equivalent permutations. Furthermore, based on the results from Section 6.2, we have that the above lower bound is tight, i.e., $\Delta_{s\to s'} = \Delta_{\text{Total}}$, when all simple permutations performed by the above heuristic involved pairs of operators with orthogonal predicates, i.e., when all permutations are very strongly equivalent.

Example 13 [Validation Sequence Improvement] Consider validation sequence $s = <o_1, o_2, o_3>$. Assume, that this sequence was used to validate dataset $D$ consisting of 1000 data elements, and that $o_1$ validated 150 ($n_1$), $o_2$ validated 100 ($n_2$), and $o_3$ validated 750 ($n_3$) of them. Thus, $\text{cost}(s, D) = 1000 + 850 + 750 = 2600$. Furthermore, let’s assume that our heuristic produced the following equivalent permutation $s' = <o_3, o_1, o_2>$ by first swapping operators $o_2$ and $o_3$, and then $o_1$ and $o_3$. According to the greedy heuristic, $\Delta_{\text{Total}} = (n_3 - n_2) + (n_3 - n_1) = (750 - 100) + (750 - 150) = 1250$. Therefore, based on the lower bound result, we are guaranteed to have $\text{cost}(s', D) \leq 1350$, i.e., the cost was cut nearly in half.

Basically, the above heuristic is traversing the permutation graph (such as depicted in Figure 2) by doing one equivalent simple permutation at a time, starting from vertex $s$. Since at every iteration we perform an equivalent simple permutation, the sequence $s'$ produced by the above heuristic remains equivalent to the original sequence $s$ (because of the transitivity of the equivalence relation). In addition, note that on every iteration we perform a simple permutation only if $\Delta_x > 0$. Because of this, we are guaranteed not to swap the same two
validation operators more than once. Therefore, the maximal number of iterations performed by the above heuristic is equal to the number of possible validation operator inversions, i.e., $k(k-1)/2$ (assuming the validation sequence has $k$ operators). However, $k(k-1)/2$ iterations are only possible when all simple permutations are permissible, which is a worst-case scenario that is far from realistic. E.g., in many real-life data mining validation scenarios the number of permissible permutations typically is $O(k)$ because of the predicate non-orthogonality between numerous pairs of operators (rule filters) that have different labels. Moreover, since the computational complexity of a single iteration is $O(k)$, i.e., dealing with $k-1$ pairs of adjacent operators, the worst case computational complexity of the heuristic is $O(k^3)$ with the typical real-life complexity being much lower, i.e., $O(k^2)$, as mentioned earlier. This is a significant improvement over the exhaustive search-based techniques that would consider all validation sequence permutations and would have worst case computational complexity of $O(k!)$.

6.4 Performance of the Greedy Heuristic

Since, as shown earlier, the sequence optimization problem is NP-hard, the greedy heuristic is not guaranteed to always produce the optimal (i.e., least costly) validation sequence because the heuristic has only polynomial computational complexity. Therefore, it is important to understand how well the proposed heuristic typically performs, i.e., how far-off it is from the optimal solution.

Unfortunately, the theoretical analysis of the heuristic performance constitutes an extremely difficult problem, because the performance of the heuristic very significantly depends on its inputs, i.e., on the specific dataset to be validated and on the initial validation sequence specified by the domain expert. In other words, given validation sequence $s$, the heuristic may be able to find an optimal permutation on one input dataset, but would produce a highly sub-optimal permutation if a different dataset is given. Conversely, given dataset $D$, the heuristic would be able to find the optimal permutation for one expert-specified sequence on $D$, but may produce only a sub-optimal permutation for a different expert-specified sequence.

**Example 14** [Optimality of Heuristic Results: Dependence on Inputs] Consider the robot-based validation at a factory, as described in Example 12. Consider the set of only two robots: robot $R_1$ labels all “red” items as good and robot $R_2$ labels
all “small” items as good. Since both robots use the same label, permutations \( s_{12} = < R_1, R_2 > \) and \( s_{21} = < R_2, R_1 > \) are equivalent (even strongly equivalent).

In general, it is easy to see that the optimal sequence depends on the input dataset: if factory manufactures more “red” items than “small” items, then \( s_{12} \) is optimal (i.e., least costly), otherwise \( s_{21} \) is the optimal one.

Furthermore, assume that \( s_{12} \) is the initial expert-specified sequence of robots and consider the following input dataset \( D \) of 100 items: 20 items are “red” and “small”, 70 - “green/small”, 10 - “green/big”. Obviously, \( R_1 \) will validate 20 of the 100 items (i.e., \( n_1 = 20 \)) and \( R_2 \) will validate 70 of the remaining 80 items (i.e., \( n_2 = 70 \)). Based on this information (i.e., \( n_1 < n_2 \)) the heuristic would swap \( R_1 \) and \( R_2 \) and would produce the optimal result in this case, since \( \text{cost}(s_{12}, D) = 100 + 80 = 180 \) and \( \text{cost}(s_{21}, D) = 100 + 10 = 110 \).

Alternatively, assume that \( s_{12} \) is still the initial expert-specified sequence of robots, but dataset \( D \) this time consists of the following 100 items: 40 items are “red” and “small”, 30 - “green/small”, 30 - “green/big”. Obviously, \( R_1 \) will validate 40 of the 100 items (i.e., \( n_1 = 40 \)) and \( R_2 \) will validate 30 of the remaining 60 items (i.e., \( n_2 = 30 \)). Based on this information (i.e., \( n_1 > n_2 \)) the heuristic would not swap \( R_1 \) and \( R_2 \). However, the optimal result would not be produced in this case, since \( \text{cost}(s_{12}, D) = 100 + 60 = 160 \) and \( \text{cost}(s_{21}, D) = 100 + 30 = 130 \).

As the above example indicates, in general it may not be possible to find any closed-form solutions describing the heuristic performance without understanding the specifics of the domain knowledge – the underlying data and the validation predicates. E.g., in the above example the understanding of the “overlap” between “small” and “red” items was more crucial than knowing the performance of individual validation operators (i.e., knowing \( n_1 \) and \( n_2 \)).

---

5As mentioned earlier, given an initial expert-specified validation sequence \( s \), the proposed greedy heuristic searches for the best sequence \( s' \) among the strongly equivalent permutations of \( s \). Note, that if we restricted our heuristic to search for the best sequence \( s' \) only among the very strongly equivalent permutations of \( s \), we would be able to accurately estimate the performance improvement, since in this case \( \Delta_{s \to s'} = \Delta_{\text{Total}} \) (as discussed earlier), i.e., the cost of the resulting sequence can be calculated solely from the performance of the individual operators in the initial validation sequence (i.e., by knowing \( n_i \)). However, the class of very strongly equivalent permutations is extremely restrictive and is not practical for most real-life applications; we have used it in this paper only for the purpose of deriving some the theoretical results. The class of strongly equivalent permutations is much broader and, therefore, gives the greedy heuristic a much larger space to work with, typically resulting in much better performance improvements.
For the same reasons, i.e., because of the dependence of heuristic performance on the domain-specific information, it is difficult to produce not only theoretical, but also simulation-based results about the performance of the heuristic, unless we choose to make oversimplifying or hugely restrictive assumptions, e.g., restrict the input data and validation predicates to be of some very limited types. Since in this paper our focus is on the general validation problem, we plan to pursue the domain-specific validation issues in our future research.

However, as discussed in Section 6.3, the proposed heuristic does provide some theoretical guarantees. As Equations (19) and (20) indicate, the performance improvement from the initial expert-specified sequence is guaranteed to be at least:

\[ \Delta_{s \rightarrow s'} \geq \sum_{i: (o_i \prec o_j) \land (o_j \prec o_i)} (n_j - n_i) \]  

In other words, the heuristic is always able to present the feedback to the user regarding the guaranteed cost savings it produces, based only on its knowledge of initial sequence performance (i.e., \( n_i \) values).

While the greedy heuristic is not guaranteed to produce optimal results in all the cases, it can generate optimal validation sequences in some circumstances. The most obvious example is when all simple permutations are very strongly equivalent, e.g., when all the predicates are pairwise orthogonal, as follows from the results in Section 6.2. In such cases, the heuristic would be able to simply sort the operators \( o_i \) in the descending order based on the number of data points (i.e., \( n_i \)) that each of the operators has validated.

Since the heuristic does not guarantee optimality, it is important to understand some of its worst-case scenarios, even without taking the domain specifics into account. Figure 3

\[
\begin{array}{cccc}
  i & p_i & l_i & n_i \\
  1 & p_1 & A & 2 \\
  4 & p_4 & A & 2 \\
  5 & p_5 & A & 1 \\
  6 & p_6 & B & 1000 \\
  \cdots & \cdots & \cdots & \cdots \\
  2 & p_6 & B & 1000 \\
  3 & p_1 & A & 2 \\
  \cdots & \cdots & \cdots & \cdots \\
  \end{array}
\]

(a) Initial sequence \((s)\)  

\[ cost(s, D) = 6025 \]

(b) Optimal sequence \((s')\)  

\[ cost(s', D) = 2037 \]

Figure 3: “Problematic” example.
illustrates one such scenario. Specifically, consider validation sequence \( s \) with six validation operators, as shown in Figure 3a. Each of the operators can label the appropriate input with an A or B label. Moreover, let’s assume that all predicates are pairwise orthogonal except for \( p_5 \) and \( p_6 \), i.e., \( p_5 \perp p_6 \). The numbers of data points (out of 1009 total data points) validated by each operator \( n_i \) are also provided in Figure 3a.

Based on our definition of sequence cost, the cost of \( s \) is equal to 6,025, i.e., sequence \( s \) had to perform 6,025 individual data checks to validate the input data. Furthermore, it is easy to see that the greedy heuristic would not perform any changes to the sequence, because \( n_{i+1} - n_i \leq 0 \) for every permissible simple permutation in \( s \). Note, that it is not permissible to swap operators 5 and 6, since neither \( l_5 = l_6 \) nor \( p_5 \perp p_6 \). However, it is easy to see that if we do not follow the greedy approach and move operator 5 all the way up (because \( p_5 \) is orthogonal to \( p_1, \ldots, p_4 \)), we can move operator 6 all the way up to the second position in the sequence (again, because \( p_6 \) is orthogonal to \( p_1, \ldots, p_4 \)), obtaining sequence \( s' \) (presented in Figure 3b) that is still equivalent to \( s \), but significantly less costly. Specifically, the cost of \( s' \) is equal to 2,037. Therefore, the greedy heuristic does not provide any performance improvements in this “worst-case” scenario, whereas an optimal solution improved performance by 3,988 operations, resulting in a cost reduction of 66.2%.

We would like to point out that such worst-case validation sequences, as shown in Figure 3a, do not occur frequently in practice. As was argued in [3], if possible, domain experts usually try to specify more general validation operators validating large amounts of data (e.g., more general rule filters in data mining applications) first, and the more specific operators later on. Therefore, it is not a very typical validation behavior to specify many very specific validation operators (i.e., \( p_1, \ldots, p_4 \)) before applying a very strong and general one (i.e., \( p_6 \)).

Note, that the proposed heuristic would be able to find the optimal sequence \( (s') \) in the above example, if we had \( n_5 = 3 \) (instead of \( n_5 = 1 \)). Clearly, the heuristic algorithm can be improved so that it can detect some of the “problematic” situations, such as the one discussed above, at the expense of increasing the computational complexity of sequence optimization. In the next section we provide a better (but more complex) heuristic that is based on the dynamic programming approach.
6.5 Dynamic Programming Approach to Sequence Optimization

Assume \( s = < o_1, \ldots, o_k > \) is an expert-specified validation sequence that is used to validate dataset \( D \). We will construct an improved validation sequence \( s' = < o'_1, \ldots, o'_k > \) as a permutation of \( s \). In the beginning, let \( s' := s \). Then, the construction of \( s' \) can be described as the following iterative process:

\begin{enumerate}
  \item for \( n := 2 \) to \( k \)
  \item \( CurrS := s'[1..n] \)
  \item \( BestS := CurrS \)
  \item \( Candidate := n - 1 \)
  \item for \( p := n \) downto \( 2 \)
    \item \( Success := False \)
    \item while \( \neg Success \) and \( Candidate \geq 1 \)
      \item \( Success := SWAP(CurrS[Candidate], CurrS[Candidate + 1, p]) \)
      \item \( Candidate := Candidate - 1 \)
    \item end while
  \item if \( \neg Success \) then exit loop
  \item if \( BestS.Cost < CurrS.Cost \) then \( BestS := CurrS \)
  \item \( Candidate := Candidate - 1 \)
  \item end for
  \item \( s'[1..n] := BestS \)
\end{enumerate}

The intuition behind this algorithm is the following. Every iteration of this algorithm (lines 2-15) tries to “optimize” a prefix of sequence \( s \) of length \( n \), where \( n = 2, \ldots, k \) (\( k \) is the length of the whole sequence). In other words, for each \( n \) the algorithm tries to optimize the sequence \( < o'_1, \ldots, o'_n > \). Clearly, we do not need to start from \( n = 1 \), since \( < o'_1 > \) is by itself optimal, i.e., no permutations are possible in the sequence of length 1. During every iteration, the optimization procedure follows the dynamic programming approach [9], i.e., the sequence of length \( n \) is optimized by taking the already “optimized” sequence of length \( n - 1 \) (the result from the previous iteration) and by trying to “insert” the \( n \)th operator (i.e., \( o'_n \)) in all permissible positions. Clearly, we have \( n \) possible such positions in the sequence of length \( n \).
More specifically, each iteration starts with the sequence of length \( n \) that is constructed from the sequence of length \( n - 1 \) (obtained from the previous iteration) by adding operator \( o'_n \) to the end of it, i.e., by keeping \( o'_n \) in its current, \( n \)th, position (line 2). Furthermore, we assume that this sequence is the best sequence of length \( n \) so far (line 3). Then we try to move the operator \( o'_n \) up through the sequence (for loop, lines 5-14) one position at a time, i.e., moving from position \( p \) to \( p - 1 \), where \( p = n, \ldots, 2 \). In order to be able to keep moving \( o'_n \) up through the sequence, some other operator has to be put in its place. The first candidate for that is \( o'_{n-1} \) (line 4), i.e., the heuristic will first try to swap \( o'_{n-1} \) and \( o'_n \). However, if at any point \( o'_{p-1} \) cannot be swapped with \( o'_p \) (i.e., because their labels are different and their predicates are not orthogonal), we then try to swap \( o'_{p-2} \) with both \( o'_{p-1} \) and \( o'_p \), and so on, increasing the block of operators to be moved up together with the \( o'_p \) as necessary (while loop, lines 7-10). If we are not able to swap any operator into the \( p \)th position, then, naturally, we are done with all permissible permutations for this iteration (line 11). On the other hand, as long as we are able to swap, we compare every new sequence against our best sequence so far,\(^6\) and update the best sequence, if needed (line 12). After we are done with all permissible permutations for this iteration, we keep the best permutation of the first \( n \) operators (line 15).

As with the greedy heuristic that was presented earlier, it is easy to see that the dynamic programming-based approach can never increase the cost of the validation sequence, only decrease it. This is the case, because in each iteration we choose the sequence that has lower cost than the default sequence, i.e., than the sequence with \( o'_n \) in its initial, \( n \)th, place.

Furthermore, the dynamic programming-based approach provides the same theoretical guarantees as the greedy heuristic described earlier. I.e., the performance improvement from the initial expert-specified sequence is guaranteed to be at least:

\[
\Delta_{s \rightarrow s'} \geq \sum_{i: (o_i \prec_s o_j) \land (o_j \prec_{s'} o_i)} (n_j - n_i) \tag{22}
\]

and the above algorithm is always able to present the feedback to the user regarding the guaranteed cost savings it produces, based only on its knowledge of initial sequence performance (i.e., \( n_i \) values).

However, the dynamic programming-based approach provides better optimization performance than the greedy heuristic. In particular, it is easy to see that it would be able to

---

\(^6\)The sequences are compared in terms of their guaranteed cost, as calculated using Equation (21).
produce the optimal validation sequence in the “problematic” example from Figure 3a, which represented the worst-case scenario for our greedy heuristic. This is the case, because in the sixth iteration (i.e., when \( n = 6 \)) the above algorithm would be able to move operators 5 and 6 up through the sequence together. In other words, it would produce the optimal result, as presented in Figure 3b.

The computational complexity of the dynamic programming-based heuristic can be estimated by noticing that each iteration (lines 2-15), i.e., when \( n = 2, \ldots, k \), has the worst case computational complexity of \( O(n^2) \). In particular, the computational complexity of the SWAP operation is proportional to the block size that is being swapped, i.e., \( p - \text{Candidate} \leq n \), and this operation cannot be attempted more than \( n - 1 \) times (based on the dynamics of \( \text{Candidate} \) variable). E.g., it can indeed take \( O(n^2) \) operations to move a block of \( n/2 \) operators through the remaining \( n/2 \) positions.

Based on the outer loop (lines 1-16), \( n \) goes from 2 to \( k \). Therefore, the overall computational complexity of the dynamic programming-based approach to sequence optimization is \( O(k^3) \), since \( 2^2 + 3^2 + \ldots + (k - 1)^2 + k^2 = O(k^3) \). While the computational complexity of the above algorithm is greater than the one of the greedy heuristic in most real-life situations, the dynamic programming-based approach produces better optimization results, as indicated earlier. However, as with the greedy heuristic, it is very difficult to derive more precise theoretical results about the performance of the above algorithm without understanding the specifics of the domain knowledge, i.e., the underlying data and the validation predicates.

7. Summary and Future Work

In this paper, we presented a general expert-driven validation approach that is based on the sequences of validation operators. We explored various properties of such sequences, such as sequence equivalence, sequence permutation and sequence optimality, and derived several theoretical results providing for a better understanding of the validation process. We also addressed the problem of optimizing the expert-specified validation sequence by searching the set of equivalent sequence permutations. This problem can also be viewed as a certain type of a scheduling problem, where costs of performing individual validation tasks highly depend on the position of this task (and other tasks) in the schedule. We showed that this optimization problem is NP-hard. In addition, we presented two algorithms – a greedy heuristic and a dynamic programming-based approach – that can be used to improve
validation performance of a given sequence.

In our approach we view validation as an entirely expert-driven process and rely on the domain expert to provide a validation sequence based on the expert’s validation decisions. Moreover, in this paper, we take the validation sequence produced by the domain expert as an exogenous input and try to transform it into an equivalent but computationally faster sequence. In other words, this paper addresses the problem of how to do the validation faster given the initial validation results, which is important for the following reasons. First, in many applications, it is necessary to re-validate data periodically because it changes over time. Since this re-validation process can be computationally expensive in some applications, it is crucial to do it efficiently, and the proposed methods address this problem. Second, the domain expert has to wait for the results of each validation step in some applications, and the generation of a more efficient validation sequence would reduce this waiting time. Third, some validation applications, such as data streams, require real-time processing. In these applications, it is necessary to produce optimized validation sequences to comply with real-time processing requirements.

This paper focused on a general approach to optimizing validation sequences and did not consider any domain-specific information. In our future work we plan to extend our general validation results in several ways. First, by considering specific application domains, such as data mining rules, data streams, and database records, we can enhance validation capabilities of the domain expert by incorporating particular domain knowledge about these domains into the validation process and letting the validation system assist the domain expert to generate better and more efficient validation sequences. For example, a significant amount of research has been done on rule-based expert systems. While the area of expert systems does not directly address the expert-driven rule validation problem, we believe that some of the results on how to deal with evolving rule bases [4] or how to improve the performance of such systems [12] may lead to the improvements of the rule validation approaches. We also believe that incorporating the domain knowledge into the validation process will not only result in better validation sequences, but also will allow to derive more precise theoretical results. Second, we did not address uncertainty as a part of the validation problem, i.e., in our approach we assume that the validation operators assign labels to validated data without any uncertainty. As an initial step, we wanted to study and understand the validation process that is fully deterministic, however, in our future work we plan to explore the possibilities to incorporate uncertainty into the validation process.
References


A. Proofs of Main Theoretical Results

Proof of Theorem 4

First, let’s assume that validation sequences $s$ and $s'$ contain validation operators $u$ and $v$ that satisfy all three conditions described above. We will show that $s \not\sim s'$.

Based on the condition 3, there exists an input element $e$ that satisfies the Boolean expression 7. Because this expression is a conjunction of several subexpressions, $e$ satisfies each of these subexpressions. Based on this we derive the following.

Since both $p_u(e)$ and $\bigwedge_{i=1}^{x-1} \neg p_i(e)$ hold, we have that $e$ satisfies predicate $p_u$ (which is at the position $x$ in $s$), but does not satisfy any of the predicates $p_1, p_2, \ldots, p_{x-1}$. These predicates are at positions $1, 2, \ldots, x-1$ respectively in sequence $s$, therefore all predicates that precede $p_u$ in the validation sequence would not match $e$. Consequently, in the sequence $s$, $e$ would be matched by predicate $p_u$ and labeled with $l_u$. Obviously, $u \prec_s v$, since otherwise $v$ (and not $u$, as we just showed) would match the input $e$ in sequence $s$.

Similarly, since both $p_v(e)$ and $\bigwedge_{j=1}^{y-1} \neg p'_j(e)$ hold, we have that $e$ satisfies predicate $p_v$ (which is at the position $y$ in $s'$), but does not satisfy any of the predicates $p'_1, p'_2, \ldots, p'_{y-1}$. Therefore, in the sequence $s'$, $e$ would be matched by operator $p_v$ and labeled with $l_v$. Obviously, $v \prec_{s'} u$, since otherwise $u$ (and not $v$, as we just showed) would match the input $e$ in sequence $s'$.

Since both $u \prec_s v$ and $v \prec_{s'} u$, condition 1 is satisfied automatically.

Based on the condition 2, $l_u \neq l_v$. Therefore, $s$ would validate $e$ differently than $s'$. Therefore, when $D = \{e\}$, we have $VI_s(D) \neq VI_{s'}(D)$. Hence, $s \not\sim s'$.
Now let’s assume that $s \not\sim s'$. We will show that these sequences contain validation operators $u$ and $v$ that satisfy all three conditions mentioned above.

$s \not\sim s' \Rightarrow (\exists D)(VI_s(D) \neq VI_{s'}(D))$. Let’s denote $VI_s(D) = (V_1, V_2, \ldots, V_{|L|})$ and $VI_{s'}(D) = (V'_1, V'_2, \ldots, V'_{|L|})$. Here $V_i$ ($i = 1, \ldots, |L|$) is a subset of $D$ labeled with the label $L_i$ by sequence $s$. Similarly, $V'_i$ ($i = 1, \ldots, |L|$) is a subset of $D$ labeled with the label $L_i$ by sequence $s'$. Since $VI_s(D) \neq VI_{s'}(D)$, we have that $(V_1, \ldots, V_{|L|}) \neq (V'_1, \ldots, V'_{|L|})$. Therefore, there exists $i$ such that $V_i \neq V'_i$.

Since $V_i \neq V'_i$, let’s assume (without loss of generality) that there exists an entity $e \in D$ such that $e \in V_i$, but $e \notin V'_i$. (It could also be $e \in V'_i$ and $e \notin V_i$, in which case the proof would be virtually the same as below.) Since $e \notin V'_i$, there exists $j \in \{1, \ldots, |L|\}$ such that $i \neq j$ and $e \in V'_j$. Note that, $e$ can not remain unvalidated by $s'$, as shown in Lemma 3.

Because $e \in V_i$, there must exist a validation operator $u = (L_i, p_u)$ in the sequence $s$ (say, at the position $x$, i.e., $pos_s(u) = x$) that validates $e$ (i.e., $p_u(e)$ is True), but none of the preceding operators do (i.e., $\neg p_i(e)$ for all $i = 1, \ldots, x - 1$). Therefore, both $p_u(e)$ and $\bigwedge_{i=1}^{x-1} \neg p_i(e)$ hold.

Similarly, because $e \in V'_j$, there must exist a validation operator $v = (L_j, p_v)$ in the sequence $s'$ (say, at the position $y$, i.e., $pos_{s'}(v) = y$) that validates $e$ (i.e., $p_v(e)$ is True), but none of the preceding operators do (i.e., $\neg p'_j(e)$ for all $j = 1, \ldots, y - 1$). Therefore, both $p_v(e)$ and $\bigwedge_{i=1}^{y-1} \neg p'_j(e)$ hold.

The previous two paragraphs combined show that condition 3 holds. Condition 2 also holds, since $u$ and $v$ operators described above have different labels (i.e., $L_i$ and $L_j$, $i \neq j$). Finally, condition 1 holds as well, because the same input element $e$ is validated by $u$ in sequence $s$ and by operator $v$ in sequence $s'$, which would be impossible when either of operators $u$ and $v$ precedes the other one in both sequences, since they both match $e$.

**Proof of Theorem 8**

Let $x$ be the largest number from the set $\{1, \ldots, k\}$, such that $o_{x+1} \prec_s o_x$. Then, let’s construct the sequence $s'' = < o''_1, \ldots, o''_k >$ as follows. Let $o''_i := o_i$, for all $i$, such that $1 \leq i \leq x - 1$ or $x + 2 \leq i \leq k$. Also, let $o''_x = o_{x+1}$ and $o''_{x+1} = o_x$.

Essentially, sequence $s''$ is the same as $s$ except for $o_x$ and $o_{x+1}$ that are swapped. Obviously, $s''$ is a simple permutation of $s$, thus $\text{dist}(s, s'') = 1$.

Now we will show that $s \sim s''$. Since $s''$ is a simple permutation of $s$, Corollary 6 gives us two conditions to be satisfied in order to have $s \sim s''$. 47
Assume $o_x$ and $o_{x+1}$ have the same label, i.e., $l_x = l_{x+1}$, then the first condition from Corollary 6 is satisfied. Therefore, $s \sim s''$. In case $o_x$ and $o_{x+1}$ do not have the same label, the only way for $s \sim s''$ to be true is for $o_x$ and $o_{x+1}$ to satisfy the second condition from Corollary 6. For the remainder of this proof we will assume that $o_x$ and $o_{x+1}$ do not have the same label, and we will show that they satisfy the second condition from Corollary 6, i.e.,

$$\neg(p_x \land p_{x+1}) \lor \bigvee_{i=1}^{x-1} p_i \lor \bigvee_{j=1}^{y-1} p_j'$$

(23)

Let’s go back to sequences $s$ and $s'$ for a moment. Since $s \sim s'$, from Corollary 5 we have that all pairs of operators from $s$, including $o_x$ and $o_{x+1}$, must satisfy at least one of the three necessary and sufficient conditions for $s \sim s'$. Let’s consider the pair $o_x$ and $o_{x+1}$.

Since $o_x$ precedes $o_{x+1}$ in $s$, but $o_{x+1}$ precedes $o_x$ in $s'$ (that’s how we chose $o_x$ in the beginning of the proof), the first condition is not satisfied by these two operators. These operators do not satisfy the second condition as well, since they do not have the same label (according to our assumption). Therefore, since $s \sim s'$, $o_x$ and $o_{x+1}$ must satisfy the third condition of Corollary 5, namely:

$$\neg(p_x \land p_{x+1}) \lor \bigvee_{i=1}^{x-1} p_i \lor \bigvee_{j=1}^{y-1} p_j'$$

(24)

where $y$ is the position of $o_{x+1}$ in $s'$. Therefore, $o_{x+1} = o_y'$. Also note, that by construction, none of $o_j' (j \in \{1, \ldots, y-1\})$ can be equal to $o_x$ or $o_{x+1}$, since $o_{x+1} = o_y'$ and $o_{x+1} \prec_{s'} o_x$.

We will show that every $o_j' (j \in \{1, \ldots, y-1\})$ is from among $o_1, \ldots, o_{x-1}$. Suppose otherwise, there exists $j \in \{1, \ldots, y-1\}$ such that $o_j' = o_z$, where $x \leq z \leq k$. Since, as mentioned above, none of $o_j' (j \in \{1, \ldots, y-1\})$ can be equal to $o_x$ or $o_{x+1}$, we can obtain an even tighter bound for $z$, i.e., $x+1 < z \leq k$.

Then, consider validation operators $o_{x+1}$ and $o_z$. $o_{x+1}$ precedes $o_z$ in $s$, because $x+1 < z$. However, $o_z$ precedes $o_{x+1}$ in $s'$, because $pos_{s'}(o_{x+1}) = y$ and $pos_{s'}(o_z) < y$. From Lemma 7 we have, that there exists $t$, $x+1 \leq t \leq z-1$, such that $o_{t+1}$ precedes $o_t$ in $s'$.

Thus, we showed that there exists $t \geq x+1 > x$, such that $o_{t+1}$ precedes $o_t$ in $s'$. However, by definition $x$ is the largest number, such that $o_{x+1}$ precedes $o_x$ in $s'$ (i.e., we chose $x$ to be the largest such number in the first paragraph of this proof). We derived a contradiction, therefore our assumption that there exists $j \in \{1, \ldots, y-1\}$ such that $o_j' = o_z$, where $x \leq z \leq k$ is incorrect. This implies that every $o_j' (j \in \{1, \ldots, y-1\})$ is
from among \(o_1, \ldots, o_{x-1}\). Therefore, every \(p'_j\) \((j \in \{1, \ldots, y-1\})\) is from among \(p_1, \ldots, p_{x-1}\). Consequently, the third necessary condition 24 of \(s \sim s'\) in this case is equivalent to:

\[
\neg (p_x \land p_{x+1}) \lor \bigvee_{i=1}^{x-1} p_i
\]  

(25)

Hence, based on the fact that \(s \sim s'\), we proved that the Boolean expression 25 holds for every element \(e \in \mathcal{E}\). However, this expression is exactly the same as the one described by Equation 23, which was needed to prove that \(s \sim s''\) (when \(o_x\) and \(o_{x+1}\) do not have the same label). Therefore, \(o_x\) and \(o_{x+1}\) satisfy the second sufficient condition of Corollary 6 and, hence, \(s \sim s''\).

Now we have \(s \sim s'\) and \(s \sim s''\). Because of the transitivity and the symmetry of the relation \(R_{\sim}\) (see Lemma 2), \(s' \sim s''\) is also true. Also, we know that \(\text{dist}(s, s') = d\) and \(\text{dist}(s, s'') = 1\). Because of how we constructed \(s''\), \(s'\) has all the same precedence inversions with respect to \(s''\) as with respect to \(s\), except for one. More specifically, \(o_x\) and \(o_{x+1}\) have the same precedence in both \(s'\) and \(s''\). Therefore, the distance between \(s'\) and \(s''\) is one less than between \(s'\) and \(s_i\), i.e., \(\text{dist}(s', s'') = d - 1\).

**Proof of Theorem 9**

\[\blacktriangleleft\]

Assume \(s \sim s'\). Let’s denote \(s_0 := s\) and \(s_d := s'\). Based on Theorem 8, there exists sequence \(s_1\), such that \(s_1\) is a safe simple permutation of \(s_0\), and also \(s_1 \sim s_d\), and \(\text{dist}(s_1, s_d) = d - 1\). Repeat this process for \(s_1\) and \(s_d\) to obtain \(s_2\), etc. In general, when we have \(s_i\), such that \(s_i \sim s_d\) and \(\text{dist}(s_i, s_d) = d - i\), we can obtain \(s_{i+1}\) (which is a safe simple permutation of \(s_i\)), such that \(s_{i+1} \sim s_d\) and \(\text{dist}(s_{i+1}, s_d) = d - i - 1\). Hence, there exists \(d + 1\) validation sequences \(s_0, s_1, \ldots, s_d\), such that \(s_0 = s, s_d = s'\), and \(s_i\) is a safe simple permutation of \(s_{i-1}\) for every \(i = 1, \ldots, d\).

Now assume, that there exists \(d + 1\) validation sequences \(s_0, s_1, \ldots, s_d\), such that \(s_i\) is a safe simple permutation of \(s_{i-1}\) for every \(i = 1, \ldots, d\). In other words, for every \(i = 1, \ldots, d, s_{i-1} \sim s_i\). By transitivity of the equivalence relation: \(s_0 \sim s_d\). Hence, \(s \sim s'\).

**Proof of Lemma 19**

\[\blacktriangleleft\]

Let’s assume \(s \cong s'\) and let’s consider an arbitrary validation operator \(o_i\) from sequence \(s\), i.e., \(\text{pos}_s(o_i) = i\). Also, let \(j = \text{pos}_{s'}(o_i)\). We have to show that \(n_i = n'_j\). We will show this by showing that \(o_i\) validates exactly the same subset of \(D\) in both \(s\) and \(s'\).

Assume otherwise, that there exists \(e \in D\) such that either (a) \(o_i\) validates \(e\) in \(s\) but not in \(s'\), or (b) \(o_i\) validates \(e\) in \(s'\) but not in \(s\). We will provide the proof for the first of these two situations. The proof for the second one is essentially identical.
Since there exists $e \in D$ such that $o_i$ validates $e$ in $s$ but not in $s'$, there must exist a validation operator $o_x$ such that $o_x \prec_{s'} o_i$ and $o_x$ validates $e$. However, $o_i \prec_s o_x$, because otherwise $o_i$ would not be able to validate $e$ in $s$ (i.e., $o_x$ would validate $e$ before $o_i$). Therefore, we have two validation operators $o_i$ and $o_x$ such that $o_i \prec_s o_x$, $o_x \prec_{s'} o_i$, and $p_i \not\perp p_x$ (since there exists $e \in D$ that can be validated by both $o_i$ and $o_x$). This is a contradiction, because by the definition of very strong equivalence all pairs of validation operators must satisfy one of two conditions (see Definition 10), whereas the pair $o_i$ and $o_x$ satisfies neither.

Therefore, given $s$ and $s'$, where $s'$ is very strongly equivalent to $s$, each validation operator validates exactly the same subset of inputs from $D$ in both $s$ and $s'$. Hence, for all $i$: $n_i = n'_j$, where $j = pos_{s'}(o_i)$. □